Doctoral Dissertation

A Study on Sublinear Progressive Algorithms and FPT Graph Algorithms

Supervisor Prof. Hiro Ito

Department of Computer and Network Engineering
Graduate School of Informatics and Engineering
The University of Electro-Communications

The degree of Doctor of Philosophy in Engineering

Kyohei Chiba

Mar. 2022

Document description

Title:

A Study on Sublinear Progressive Algorithms and FPT Graph Algorithms

Title (in Japanese):

劣線形漸進型アルゴリズムとグラフ上の FPT アルゴリズムの研究

Author

Kyohei Chiba

Approved by the Dissertation Committee:

Chairperson: Prof. Hiro Ito

Member: Prof. Satoru Kobayashi

Member: Prof. Yoshio Okamoto

Member: Associate Prof. Yasuhiko Takenaga

Member: Associate Prof. Shinnosuke Seki

劣線形漸進型アルゴリズムとグラフ上のFPTアルゴリズムの研究

千葉 恭平

内容梗概

アルゴリズム研究者の伝統的な基本合意事項として、多項式時間アルゴリズムが存在する問題は効率的に解けるとされてきた。有用な応用を持つ問題の多くがNP困難問題であり、P ≠ NPの前提の元では多項式時間アルゴリズムは存在しないので、多項式時間の近似アルゴリズムが多く開発されてきた。しかし近年の計算機の性能の飛躍的な向上に伴って、扱うデータの量が古い常識では考えられないほど大規模になり、データの全てを読み込むだけでもかなりの時間を必要とするようになってきた。そのため多項式時間アルゴリズムを用いたとしても現実的には膨大な計算時間を要することがあり、従来の計算モデルからのパラダイムシフトが起きている。我々はこのパラダイムシフトを「劣線形時間パラダイム」と呼び、ビックデータの解析手法として、線形時間アルゴリズムや劣線形時間アルゴリズムを開発することの重要性が今後ますます高まっていくと予想している。この問題に対処するため、我々は二つの技法に取り組んだ。

一つ目は劣線形漸進型アルゴリズムである。まず、厳密アルゴリズムとは入力を全て読み込んだ上で厳密解を出力するアルゴリズムである。次に、劣線形時間アルゴリズムとはデータの一部を読み込むことで解を出力する確率的アルゴリズムで、今世紀に入ってから盛んに研究されるようになり、これまでに数多くの研究が存在するアルゴリズムである。そして、漸進型アルゴリズムとは多項式時間近似アルゴリズムを用いて、最初は少ない計算時間から粗い近似解を求め、計算時間の増加に伴って段々と精度の良い近似解を出力するアルゴリズムである。既存の漸進型アルゴリズムの研究は、個別の問題を具体的に解く多項式時間近似アルゴリズムを漸進型化するものであり、確率的アルゴリズムや劣線形時間アルゴリズムを考慮した研究は存在せず、一般的な変換法を示した成果は無い。この漸進型アルゴリズムを劣線形時間にまで拡張したものを劣線形漸進型アルゴリズムと呼ぶ。多項式時間アルゴリズムは入力をすべて読みこんでから動作することを前提としているが、劣線形漸進型アルゴリズムは徐々に読み込むデータ量を増やしながら結果の精度を段々と上げていくという手法であるため、大規模データに対してもデータの読み込み量に応じた精度の解を

出力することができる。我々は任意の劣線形時間アルゴリズムと厳密アルゴリズムの組に対して劣線形時間アルゴリズムを構成できることを証明し、劣線形漸進型アルゴリズムの枠組みと理論基盤を構成した。

二つ目は固定パラメータ容易（FPT）アルゴリズムである。FPT アルゴリズムとは計算時間があるパラメータに対してのみ指数的であり、入力の大きさに対しては多項式的であるようなアルゴリズムである。FPT アルゴリズムはパラメータの値が小さい場合に問題が効率的に解けるため、入力が大きい場合も問題を扱いやすくなっている。我々は実験計画法に重要な応用を持つグラフ理論の組合せ最適化問題、具体的には与えられたグラフの全ての辺を最小数のクリークで被覆する問題を扱った。クリークとは誘導部分グラフが完全グラフであるような頂点集合である。この時、クリーク同士の辺上での重なり合いを許し、さらに元のグラフからの食み出しも許すような被覆を考える。この問題は有限射影平面やブロックデザインやスクールガールの問題の自然な拡張になっており、食み出しを考慮することがこれらの応用に対して重要な意味を持つ。例えば n 個のアイテムをいくつかの実験試行で比較したいとき、同時に比較できるアイテムは最大でも m 個で、比較しなければならないアイテムの組が与えられている場合、最小の試行回数を見つけることがこの問題として形式化される。この際、比較されても比較されなくても良い組が食み出しに相当している。我々はこの問題を広義辺被覆と名付けた。既存研究では、食み出しを考慮した結果は存在しない。本研究では $k = 3$ の場合について考え、広義 K_3 辺被覆問題が NP 完全であることを証明し、二種類の FPT アルゴリズムを構成した。一つ目は与えられたグラフ $G = (V, E)$ に対して、G に含まれる K_3 の数を k 個した場合の $O(|V||E| + 2k|E|)$ 時間アルゴリズムであり、もう一つは、木幅 t の木分解が与えられた場合の $O(2\{2(t + 1)(t + 2)\}t2n)$ 時間のアルゴリズムである。

A Study on Sublinear Progressive Algorithms and FPT Graph Algorithms

The degree of Doctor of Philosophy in Engineering Kyohei Chiba

Abstract

Traditionally, for a given problem, if a polynomial-time algorithm exists, the problem is described as being "efficiently" solvable. For this reason, polynomial-time algorithms have been developed for many problems. Moreover, for NP-hard problems, which are believed to be unsolvable in polynomial time, our practical goal has been to find polynomial-time approximation algorithms. In recent years, however, as the performance of computers has increased drastically, the amount of data has become unimaginably large, and a paradigm shift from the traditional computational model has occurred. This is because the amount of data is so enormous that it takes a lot of time even just to read the whole data. We call this paradigm shift the "sublinear computation paradigm," and expect that it will become increasingly important to develop linear and sublinear-time algorithms. To address this problem, we have worked on two techniques in this research.

The first technique is the sublinear progressive algorithms. A sublinear-time algorithm is an algorithm that outputs a solution by reading a portion of the data. The idea was given in the 1990s and many research work have been done particularly in this century. A progressive algorithm outputs an approximate solution in a short time, and then gradually improves the accuracy of the results as time progresses. We extend this idea to sublinear-time algorithms and we call them sublinear progressive algorithms. A sublinear progressive algorithm is an algorithm that outputs a solution, by reading a small amount of data at first, updates the accuracy of the solution with every reading of the input, and finally outputs the exact solution. A polynomial-time algorithm, on the contrary, is basically supposed to work after reading all the inputs. A sublinear progressive algorithm, however, outputs some solutions with reading only a constant-sized portion of the input, and hence it is expected to output solutions in a very short time even for large data. Although some progressive algorithms have been known in the context of polynomial-time algorithms, they are polynomial-time algorithms that solve individual problems, and there is no research that considers progressive algorithms for probabilistic or sublinear-time algorithms. We give a

theoretical framework of sublinear progressive algorithms and present *Sublinear Progressive Algorithm Theory* (*SPA Theory*, for short), which enables us to make a sublinear progressive algorithm for any property that has both a constant-time algorithm and an exact algorithm (an exponential time one is allowed) without losing any computation time in the big-O sense. This provides a theoretical foundation for sublinear progressive algorithms.

The second technique is fixed-parameter tractable (FPT) algorithms. An FPT algorithm runs in exponential time only for some parameters, and polynomial for the size of the input. This algorithm solves the problem efficiently for small values of the parameters and makes the problem easier to handle even for large inputs. In this research, we construct an FPT algorithm for a combinatorial optimization problem that has important applications in the design of experiments. Especially, we consider a problem of covering the edges of a given graph with a minimum number of cliques. We allow cliques to overlap and the "spilling-out" of a clique from the edges of the graph. We call this problem the K_k *edge cover problem in a wide sense*. This problem is a common extension of block design and schoolgirl problems. Allowing for spilling-out is useful for those applications. For example, suppose that we would like to compare n items in multiple experimental trials, the maximum number of items that can be compared simultaneously is k, and the pairs of items that must be compared are given by a graph. In this case, finding the minimum number of trials is formalized as this problem. In the previous researches, there are many results that consider problems of covering vertices and edges with a minimum number of cliques. However, there are no theoretical results that take spilling-out into account. In this research, we consider the case of $k = 3$, prove that it is NP-complete, and construct an FPT algorithm: (1) for a given graph $G = (V, E)$, where k is the number of K_3s in G, there exists an $O(|V||E| + 2^k|E|)$-time algorithm, and (2) given a tree-decomposition of width t, there is an $O(2^{2(t+1)(t+2)}t^2|V|)$-time algorithm.

A Study on Sublinear Progressive Algorithms and FPT Graph Algorithms

Contents

Acknowledgments

The five years I spent as a graduate student at the University of Electro-Communications have been a precious experience in my life. I would like to thank all the professors, staff, and students who supported me during my time at the university.

My deepest gratitude goes to Prof. Hiro Ito. He led me to the field of theoretical computer science and introduce discrete mathematics to me. He has always provided me with fun and stimulating discussions and suggestions, and has not only given me good advice on my research difficulties but also has given me many valuable words on how to be a researcher. He also gave me the opportunity to stay at Haifa University. I sincerely appreciate him for providing me with six years of research guidance in completing this dissertation.

Prof. Satoru Kobayashi, Prof. Yoshio Okamoto, Associate Prof. Yasuhiko Takenaga and Associate Prof. Shinnosuke Seki, who are referees of this doctoral dissertation together with Prof. Ito, gave me valuable suggestions and polite guidance. I would like to express my gratitude to them for their comments and guidance in the preparation of this dissertation.

I truly thank Dr. Rémy Belmonte of Université Paris-Dauphine for his advice and suggestions through our daily discussions and presentations at research meetings.

I am also grateful to Prof. Ilan Newman, my supervisor during my stay at the University of Haifa. He was very welcoming to me as a visitor and I was able to advance my research on sublinear progressive algorithms through discussions with him.

I would also like to express my gratitude to all my co-authors of our paper: Assistant Prof. Michael Lampis, Assistant Prof. Atsuki Nagao, and Associate Prof. Yota Otachi. Without their help, I would not have been able to complete our paper.

In addition, I would like to thank all the members of Ito Laboratory who have helped me in my daily activities in the laboratory.

Lastly, I would like to sincerely thank my family for supporting selflessly my research at the University of Electro-Communications.

Kyohei Chiba

Tokyo, December 3, 2021

Chapter 1 Introduction

As a traditional basic consensus among algorithm researchers, linear-time algorithms have been regarded as the fastest algorithms. However, with the remarkable development of information systems and database technologies, we face a paradigm shift from the traditional computational models because of the increasing amount of data that should be handled. It is called the "*sublinear computation paradigm*," which was proposed in the academic research project "Foundations of Innovative Algorithms for Big Data (ABD14)" in Japan [94]. Even polynomial time algorithms may take a huge amount of time in reality when dealing with big data that was unthinkable according to old common sense. Hence, in computer science, the study of analysis methods for big data is one of the most important issues.

1.1 Property Testing and Sublinear-Time Algorithms

Algorithms that work by reading all of the data are necessarily polynomial time or superpolynomial time algorithms. Certainly, it is necessary to read all of the input for calculating an exact solution, but we have cases where it is not necessary to read all of the input for getting an approximate solution. If the exact solution can be obtained by reading part of the input, then there is waste in the input representation. From now on, we assume that there is no waste in the input representation. Various approximation algorithms have been studied, and among them, *sublinear-time algorithms*, which compute by reading only a part of the data, and in particular *constant-time algorithms* which only look at a constant-sized part of the data no matter how large the data is, are beginning to attract attention. Since these algorithms do not read all of the input, they are necessarily probabilistic algorithms, and the solutions may contain errors. In other words, it is an attempt to obtain a solution from a part of data by allowing two kinds of ambiguity: approximation parameter and fault probability. The most studied framework in sublinear-time algorithms is *property testing*. Property testing is a relaxation of the decision problem, which probabilistically outputs a yes-no answer as to whether a given input satisfies the desired property or is far from the property.

Property testing was initially presented by Blum, Ludy, and Rubinfeld [22]. They constructed algorithms to test monotonicity, linearity, and other properties of

functions. Later, Rubinfeld and Sudan formulated the concept of property testing [96]. They introduced the idea of the distance between an input function and the property, and constructed algorithms to test whether the input satisfies the property or is far from the property. They conducted research on algebraic properties such as multi-linearity and low-degree polynomials. In addition, Alon et al. conducted a property testing of graphs using Szemerédi's regularity lemma [9]. They used Szemerédi's regularity lemma to obtain a polynomial time algorithm that find a subgraph that is not k-colorable or k-colorable by changing (adding or deleting) at most ϵn^2 ($\epsilon > 0$) edges on $n-$vertex graph [104]. As an extension of this work, Goldreich, Goldwasser, and Ron made algorithms for property testing of graphs [59]. They constructed a framework on dense graph models for graphs with many edges, i.e., the adjacency matrix representation is valid. They introduced the notion of distance between the input graph and the property, that is, the input is ϵ-far from the property if the changing (addition or delection) of ϵn^2 edges in the input graph does not satisfy the property. They presented constant-time algorithms for k-coloring, ρ-clique, ρ-cut, and ρ-bisection. In this dense graph model, it is proved that monotone property[1] [13] and hereditary property[2] [12] property are constant-time testable. Later, these proofs were extended and the necessary and sufficient conditions for testable properties in the dense graph model were clarified [11].

On the other hand, there are some properties that are often established in sparse graphs such as connectivity and planarity. In dense graphs, the graph is almost always connected and non-planar. In order to consider these properties, another model handling sparse graphs has been introduced. This graph model was introduced and examined by Goldreich and Ron as the bounded degree model [60]. It has been proved that cycle-freeness [60], k-edge connectivity [110], and minor-closedness[3] [20] are constant-time testable. Regarding property testing in specific classes of graphs, it has been proved that arbitrary properties can be tested in constant time for forests [80], outerplanar graphs [15], hyperfinite graphs (see Definition 23 in

[1] A property of a graph is called monotone if it is closed under removal of edges and vertices.

[2] A property of a graph is called hereditary if it is closed with respect to induced subgraphs.

[3] A property of a graph is called minor-closed if it is closed under removal of edges/vertices and edge contraction.

Chapter 2) [89], and hierarchical scale-free graphs (see Definition 31 in Chapter 2) [66].

1.2 Progressive Algorithms

A *progressive algorithm* outputs a tentative solution in the middle of calculating the exact solution of the problem, and updates the accuracy of the solution. On the other hand, an *exact algorithm* outputs the exact solution of a problem. For some problems, exact algorithms require much time (e.g. exponential time) and we may need faster approximation algorithms. Even if the exact algorithm runs in polynomial time, we sometimes need faster ones. For example, let us consider the case when we need to immediately respond to a sudden request. Specifically, route searching problem and emergency evacuation planning problems, there are often situations where the situation changes from time to time and we want to find an approximate solution immediately, even if it is not the optimal solution. You may not respond immediately if you use exact algorithms to deal with big data. You want to get an immediate solution, even if it is an approximate result. If you can get an approximate solution immediately, you will be able to meet the request. After that, if you have extra time, then it is desirable to take time to find a better solution and finally the best one. Ideally, a method can increase accuracy gradually as time elapses. This means that the computation time to output each approximate solution get longer and longer with each output. This is the *progressive algorithm*, specifically, it is capable of outputting a sequence of solutions satisfying a monotonically decreasing function.

Progressive algorithms were proposed in the application form of ABD14. Later, the same term, *progressive geometric algorithms*, was proposed by Alewijnse et al. at SoCG 2014 [8]. They showed how to combine the $(1 + \epsilon)$-approximation algorithm with the exact algorithm to obtain an efficient progressive algorithm. In particular, they gave a progressive geometric algorithm that computes convex hulls and computes popular places from trajectory data. After this, progress algorithms were proposed for problems including. group Steiner tree search [81], sorting in the external memory model [84], Euclidean minimum spanning tree [85], Huffman coding and the improvement of convex hulls and its extension to 3D [44], the closest pair problem [86], and the weighted interval scheduling problem [95]. All of them require polynomial

time even to output the first approximation solution. This is because these algorithms read all of the input at the first step of the algorithm.

There is no research on progressive algorithms for probabilistic algorithms and sublinear-time algorithms before our research. It is a natural idea to consider a progressive algorithm that outputs a solution by reading a part of the input. Then we can say that the algorithm is progressive if it outputs solutions whose accuracy becomes better and better according to the amount of the input read by the algorithm. We call such algorithms *sublinear progressive algorithms* [39].

1.3 Sublinear Progressive Algorithms

A sublinear progressive algorithm is an algorithm that initially computes an approximate solution from a small amount of data, and gradually updates the accuracy of the solution by reading more data. Although this idea may be considered to be similar to the known progressive algorithms, it is not true. Because all of the known progressive algorithms read the whole input at first, that is, they are based on the techniques of linear or superlinear algorithms. On the other hand, our sublinear progressive algorithms use sublinear-time algorithms, which read only sublinear-sized part of the input. For this reason, it is not possible to directly use the techniques presented for the known progressive algorithms and a completely new framework and theorems are required to establish the theory of sublinear progressive algorithms. We start with explanation how to formulate sublinear progressive algorithms. In addition, we explain some other known ideas similar to sublinear progressive algorithms and explain the differences between them. We introduce a brief overview of the methods resulting from these considerations. The method of constructing a sublinear progressive algorithm in this dissertation is a technique of constructing a progressive algorithm by combining a constant-time algorithm and an exact algorithm. The sublinear progressive algorithms presented in this work satisfy the following conditions.

- The computation time to output a solution with a given accuracy is a constant multiple of the computation time of the original constant-time algorithm.
- The total computation time is a constant multiple of the computation time of the original exact algorithm.

This means that the computation time loss of the sublinear progressive algorithm

compared to the original algorithms is at most a constant multiple. We prove that there exists an ideal sublinear progressive algorithm that satisfies these conditions.

1.4 FPT Graph Algorithms

No polynomial time algorithm is known for any NP-complete of NP-hard problem yet. The problem of determining whether it is possible to solve these problems in polynomial time is called the P vs. NP problem and is one of the fundamental unsolved problems in computer science. A known exact algorithm to compute NP-complete or NP-hard problems requires exponential time. However, there are problems that can be computed in time polynomial in the size of an input but exponential in some parameter k. Then, if k is small, those problems can be solved in polynomial time for the input size. Such an algorithm is called a fixed-parameter tractable (FPT) algorithm because it can solve the problem efficiently when the parameters are fixed to small values. The first systematic study of parameterized computational complexity was done by Downey and Fellows [47]. For example, the most well-studied problem in this area is the *vertex cover problem*. A vertex cover of a graph is a set of vertices that includes at least one endpoint of every edge of the graph.

Problem Vertex Cover

Instance: A graph $G = (V, E)$, a positive integer $k \geq 1$.

Question: Does G have a vertex cover of size at most k?

It is well known that the vertex cover problem is NP-complete. For this problem, many FPT algorithms have been developed, and these algorithms have been improved and speeded up. The computation time of the best algorithm is $O(kn + 1.28^k)$ [17, 31, 34, 35, 91]. In terms of parameterized computational complexity, enumeration [48] and kernelization [1] of solutions have also been studied. Kernelization of the problem and depth-bounded search trees are used to speed up FPT algorithm [90]. The problem has been studied on planar graphs [5, 7, 28]. An FPT algorithm that works fast enough for practical use with large inputs is developed and experimentally evaluated [2, 6, 33].

Since each combinatorial optimization problem has a very different setting, a specialized FPT algorithm is required for each problem. Hence, it is unlikely that the existing methods can be applied directly, and new problems require their own FPT

algorithms. However, it is important to note that not every problem in NP has an FPT algorithm. For example, there is an $O(2^k n^2)$ algorithm for vertex cover, but there is no $O(n^{o(k)})$ algorithm for independent set under the assumption that FPT $\neq W[1]$. Keeping this in mind, we have considered two FPT algorithms for a combinatorial optimization problem that has important applications in *Design of Experiments*.

1.5 Design of Experiments

Design of experiments is the study (field) of designing efficient experimental methods. It is closely related to school girl problem [78], block design [29, 97, 106], and graph theory [45]. In particular, the application of block design to design of experiments has been well studied [42, 102, 103]. We consider the following problem.

Problem 1. There are several samples and machines. Each sample can only be evaluated relatively. The number of samples that machines can hold is fixed. You have to compare several combinations several times. If the number of samples is a, the number of machines is b, the number of samples the machine can hold is c, and the required number of comparisons between each pair of items is d, what combination minimizes the number of experiments?

This is one of the problems in the design of experiments and the block design problem. This combinatorial optimization problem can be solved for a small number of constant pairs, but as the number increases, it becomes difficult to find a solution. No general solution or equation have been found. In graph theory, this problem can be thought of as the problem of covering edges of a complete graph with *cliques*. [1] In an actual comparison experiment, however, there may be don't-care pairs that are allowed not to be compared. This corresponds to the problem of allowed to cover pairs of vertices that do not have edges in the input graph, and this situation can be represented by a "spilling-out" of cliques in the input graph. This is a very easy extension, but no studies have been done to consider the "spilling-out" in the covering. We call this problem the K_k *edge cover problem in a wide sense* and investigate the computational complexity and FPT algorithms of this problem. This problem can be regarded as a common generalization of the design of experiments and block design.

[1] A clique is a subset of vertices of a graph such that every two vertices are adjacent.

1.6 Our contributions

Sublinear Progressive Algorithms

The main purpose of this research is to develop a framework and theoretical basis for sublinear progressive algorithms and to provide a theoretical proof for the computation time and the accuracy of the solution. The sublinear progressive algorithm constructed by the method in this dissertation is briefly described as follows. If there exist a sublinear-time algorithm and an exact algorithm for a property, we can construct a sublinear progressive algorithm, that is, given a number of outputs $r \geq 2$ and a time $t \geq 1$ to get the first solution, the algorithm outputs r solutions $\{S_1, S_2, \ldots, S_r\}$ (S_r is the exact solution) in sequence, satisfying the following conditions.

(1) S_1 is obtained in time $O(t)$.

(2) for any $i \in \{1, \ldots, r\}$ the worst-case computation time T_i for getting S_i is $O(T^*(S_i))$, where $T^*(S_i)$ is the computation time to get S_i by using the original constant-time algorithm (if $i \in \{1, \ldots, r-1\}$) or the exact algorithm (if $i = r$).

(3) $R := \max_{i \in \{2,\ldots,r\}} T_i/T_{i-1}$ is minimized in the big-O sense. It means that the maximum and minimum values are at most a constant multiple.

(4) The ϵ_i and p_i are not the input, but the $epsilon_i$ and p_i of the solution output by the algorithm.

This means that the time to output $\{S_1, \ldots, S_{r-1}\}$ (resp., S_r), using the original sublinear-time algorithm (resp., the exact algorithm) from the beginning and the time to output $\{S_1, \ldots, S_{r-1}\}$ (resp., S_r) using the sublinear progressive algorithm are the same in the big-O sense. This result enables the conversion of any sublinear-time algorithm and any exact algorithm into a progressive algorithm. This bridges the gap in computation time that existed between polynomial time algorithms and sublinear-time algorithms and makes it possible to smoothly switch between algorithms. In addition, we show the relationship between approximation parameter and fault probability for a given computation time. We clarify the relationship between computation time and accuracy, and it is possible to run the algorithm up to the desired time-based accuracy, which will improve the usefulness of sublinear-time algorithms.

FPT Graph Algorithms

We formulate the K_k edge cover problem in a wide sense, which has important

applications to design of experiments and block design problems [37]. This is the problem of edge covering by cliques in graph theory. The main purpose of this research is to consider the "spilling-out" that was not considered in the past research and we clarify the computational complexity of this problem. We start with small k, because it contains an existing problem. A brief overview of the formulation is as follows. For a given graph $G = (V, E)$, we find the minimum number of 3-cliques (K_3s) that cover all edges of G. The minimum size of a K_3 edge cover is denoted by $\gamma_3(G)$. Multiple covering or covering one edge by more than one 3-clique is allowed. Moreover, in this problem, we allow "spilling-out," i.e., a set of three vertices $\{x, y, z\}$ can be covered by a 3-clique even if the induced subgraph by them is not a clique. We call this problem the K_3 *edge cover problem in a wide sense*. The problem is defined as follows.

Problem K_3-EDGE-COVER-IN-A-WIDE-SENSE (K_3EC)

Instance: A graph $G = (V, E)$, and a positive integer $h \geq 1$.

Question: $\gamma_3(G) \leq h$?

For this problem we obtain the following results. Let C_4 and C_5 be the cycles of length 4 and 5, respectively.

Theorem 2. *K_3EC is NP-complete even if graphs are restricted to planar, cubic, and C_4, C_5-free as subgraphs (i.e., not restricted to induced ones).*

The proof is done by a reduction from the maximum independent set problem on planar and cubic graphs. Since it is NP-complete, we constracted two FPT algorithms as follows.

Theorem 3. *For K_3EC, there is an $O(|E||V| + 2^k|E|)$-time algorithm, where k is the number of 3-cliques in G.*

Theorem 4. *For K_3EC, if a tree-decomposition of width t is given, there is an $O(2^{2(t+1)(t+2)}t^2|V|)$-time algorithm.*

Chapter 2 Formulation of Sublinear-time Algorithms for Sublinear Progressive Algorithms

2.1 Introduction

There is a lot of research being done using big data, for instance, evacuation plan problems, protein function predictions by combinatorial rigidity, data structure, sparse structure extraction, and data clustering theory using a bayesian approach. Requirements for computer algorithms are also evolving, particularly in the need for speed. For example, in the past, polynomial-time algorithms were considered fast, but if we applied an $O(n^2)$-time algorithm on big data of a peta-byte scale or more, we would have encountered problems with computational resources or the running time. When we handle big data, even a linear-time algorithm may be too slow. Certainly, in the era of big data, we need sublinear-time algorithms. If the computation time of an algorithm is $o(n)$, where n is the size of an input, then the algorithm is called a sublinear-time algorithm. If the running time is constant (i.e., $O(1)$), then it is called a constant-time algorithm, which is a special case of a sublinear-time algorithm.

This paradigm shift from traditional computation models is called the “sublinear computation paradigm,” which was proposed in the academic research project "Foundations of Innovative Algorithms for Big Data (ABD14) [94]" in Japan. Under this paradigm, we have obtained many fruitful results [76, 77], especially the area of constant-time algorithms. The paper [66] presents a constant-time “universal” tester for a model of complex networks, and was selected as one of the best three work in the final report of the project.

In this section, we survey sublinear-time, mainly constant-time, algorithms. In this area, property testing is the most examined framework. Property testing probabilistically distinguishes between an input having a predetermined property from that the input is far from satisfying the property. Property testing was first presented by Rubinfeld and Sudan [96] in 1992 in the context of program checking. The first study that presented the notion of constant-time testability of combinatorial structures (mainly graphs) was given by Goldreich, Goldwasser, and Ron [59], whose conference version appeared in 1995 (STOC’95). Many studies following their idea of testability have appeared and the importance of this area is growing. See [21, 57, 58]

for details.

This section is organized as follows. After presenting the basic notation and terminology in Subsection 2.2, we show some tools useful in this area in Subsection 2.3. Basic techniques are explained by examples based on fundamental problems in Subsection 2.4. Next, we present key results in this area, focusing mainly on characterizations of constant-time testable properties in Subsection 2.5.

2.2 Notations and Terminology

Basic terms and symbols

In this section, a graph is simple, i.e., it has neither self-loop nor parallel edges, unless otherwise stated. For simplicity, we omit rounding operators necessary to ensure that all values of formulas such as $\sqrt{n}$ are integers.

Let $\mathbb{Z}$ and $\mathbb{R}$ be the set of integers and real numbers, respectively. For any set of real numbers R, $R^+ := \{x \in R \mid x > 0\}$ and $R_0^+ := \{x \in R \mid x \geq 0\}$, e.g., $\mathbb{Z}_0^+$ is the set of nonnegative integers.

Oracles

A sublinear computation time means that an algorithm does not read the whole data of an input except for the case where an input is very small (i.e., smaller than a constant). Thus in other to consider sublinear-time algorithms, how to model the problems is important. A sublinear-time algorithm gets the data of the input through an oracle. If an algorithm makes a query, then an oracle gives a constant-sized answer. For example, in the dense-graph model, which is one of the most studied models, the *edge-oracle* is used: if an algorithm asks a pair of vertex ID's, say (i, j) then the oracle answers 1 if there is an edge between them, and 0 otherwise. Here, we normally assume that the ID of vertices are given by a set of successive positive integers from 1 to n, where n is the number of vertices and the algorithm knows n. Through this oracle, algorithms get (partial) information of the input graph.

Since an algorithm reads only a part of the input, getting a correct result is basically impossible. Thus we should introduce a relaxation. We explain "property testing," which is the most studied framework in subliner-time algorithms.

In this framework, we allow an approximation error: An algorithm for testing a property accepts an input with high probability (say more than $2/3$) if the input has

the property, or rejects the input with high probability if it is far from having the property. To treat this idea mathematically, we must define what a property is and what being "far" means.

Distance and ϵ-farness

The above "far" is quantified by using a positive real number $\epsilon > 0$: In order to explain the farness, we use graphs as examples. For other types of inputs, e.g., functions, grammars, strings, images, and figures, similar methods as graphs are used.

We introduce the distance between two instances (= inputs). We can define the distance only between two instances of the same size, say N. Note that since an input is represented by the answers of an oracle, N is equal to the number of possible different queries, e.g., for the edge-oracle, $N = n^2$, where n is the number of vertices. [1] In other words, if the inputs are graphs, then we define the distance only between graphs with the same number of vertices. We call a graph that consists of n vertices an *n-graph*.

The distance between two instances I and I' of size N, denoted by $\mathrm{dist}(I, I')$, is defined as follows. Let $\ell(I, I')$ be the number of distinct queries of the oracle whose answers are different between I and I', e.g., if $I = G = (V, E)$ and $I' = G' = (V, E')$ are n-graphs and the oracle is the edge-oracle, then $\ell(G, G')$ is the number of pairs $(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, n\}$ such that $(i, j) \in E \wedge (i, j) \notin E'$ or $(i, j) \notin E \wedge (i, j) \in E'$. See Fig. 1 for an example: $\ell(G, G') = 2$ since removing $(1, 4)$ and adding $(4, 6)$ are necessary to make G equal to G'.

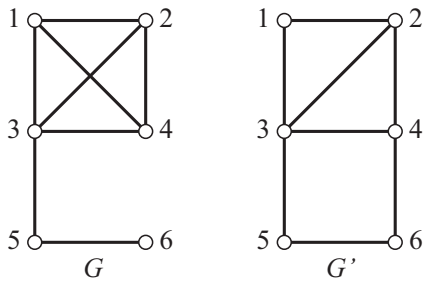


Figure 1: $\ell(G, G') = 2$.

[1] If the input is an undirected simple graph, then it must be $\binom{n}{2} = n(n-1)/2$. We normally use, however, n^2 for simplicity. Note that multiplying any constant has essentially no effect.

Now,

$$\mathrm{dist}(I, I') := \frac{\ell(I, I')}{N}. \tag{1}$$

Next, we define the distance between an instance and a property. Before showing the definition, we define properties. Let Π be the universal set of possible instances. Let Π_N be the subset of instances of Π with size N. Clearly $\Pi = \bigcup_{i=1}^{\infty} \Pi_i$. A *property* $\mathcal{P}$ of graphs is closed under isomorphism if $G \in P$ and G and G' is isomorphic, $G' \in P$ as well. The intuitive meaning of two instances being isomorphic is that they are the same except for their labels (IDs), e.g., when we consider graphs, two n-graphs $G = (V, E)$ and $G' = (V', E')$ are *isomorphic* if there is a bijection $\pi : V \to V'$ such that $(i, j) \in E \Leftrightarrow (\pi(i), \pi(j)) \in E'$. For an example of properties, a graph property "planar" is defined by the set of planar graphs. Note that this is closed under isomorphism. For any property $\mathcal{P}$, $\mathcal{P}_i := \mathcal{P} \cap \Pi_i$.

The distance between an instance I and a property $\mathcal{P}$ is defined as follows. Let N be the size of I. Then,

$$\mathrm{dist}(I, \mathcal{P}) := \begin{cases} \min_{I' \in \mathcal{P}_N} \mathrm{dist}(I, I') & \text{if } \mathcal{P}_N \neq \emptyset, \\ \infty & \text{otherwise.} \end{cases}$$

Let I and I' be the instances and $\mathcal{P}$ be a property. For a positive real number $\epsilon > 0$, we say that I and I' are ϵ-*far* if $\mathrm{dist}(I, I') > \epsilon$; otherwise, ϵ-*close*. We say that I and $\mathcal{P}$ are ϵ-*far* if $\mathrm{dist}(I, \mathcal{P}) > \epsilon$; otherwise, ϵ-*close*.

Testers

We define testing algorithms, one-sided error, query complexity, and testers as follows.

Definition 5. A *testing algorithm* for a property $\mathcal{P}$ is an algorithm that, given query access (by the oracles) to an instance I, accepts every graph from $\mathcal{P}$ with probability at least $2/3$, and rejects every graph that is ϵ-far from $\mathcal{P}$ with probability at least $2/3$. If the testing algorithm accepts every graph from $\mathcal{P}$ with probability 1, then the algorithm is called *one-sided-error*.

The success probability $2/3$ may look too small to apply to actual situations. However, this is not essential, since we can decrease fault probability to be any small positive value by iterating the algorithm a constant number of times that depends on the value.

Definition 6. (Query complexity and tester) The number of queries made by an algorithm to the given oracle is called the *query complexity* of the algorithm. If the query complexity of a testing algorithm is bounded by a constant independent of the size of instance N (but it may depend on ϵ), then the algorithm is called a *tester*. A property is *testable* if there is a tester for the property.

2.3 General Tools

Inequalities useful for bounding probabilities

First, we present some general tools useful for analyzing the probabilities of randomized algorithms. Since these tools will be used later in this section, readers can skip this section and go to the next section for the nonce, and return when they appear later.

Lemma 7. *For any real value x, $1 + x \leq e^x$.*

Proof: It is easily obtained by differentiating $e^x - x - 1$ and $e^0 = 1$. □

Theorem 8. [64] (Hoeffding's inequality) *Let $X_1, \ldots, X_s$ be independent random variables bounded as $a_i \leq X_i \leq b_i (a_i < b_i)$ for all $i \in \{1, \ldots, s\}$. $\overline{X} := \frac{1}{s}\sum_{i=1}^{s} X_i$. Let $\mathrm{Ex}[\overline{X}]$ be the expected value of $\overline{X}$. Then for any $t \geq 0$, the probability that $|\overline{X} - \mathrm{Ex}[\overline{X}]| \geq t$ occurs is bounded by the following inequality:*

$$\Pr[|\overline{X} - \mathrm{Ex}[\overline{X}]| \geq t] \leq 2\exp\left(-\frac{2s^2t^2}{\sum_{i=1}^{s}(b_i - a_i)^2}\right). \tag{2}$$

The regularity lemma

Next, we present the monumental lemma known as Szeméredi's regularity lemma. Before explaining this lemma, we need to provide some terms.

For a pair of subsets of vertices $A, B \subseteq V$ of a graph $G = (V, E)$, we denote the set of edges between A and B by $E(A, B)$, i.e., $E(A, B) := \{(v, w) \in E \mid v \in A, w \in B\}$. The *density* between A and B is defined as $\mathrm{den}(A, B) := \frac{|E(A,B)|}{|A||B|}$.

Definition 9. (ϵ-regular pair) Let $0 < \epsilon \leq 1$ be a real number and $A, B \subseteq V$. A pair (A, B) is called *ϵ-regular* if $|\mathrm{den}(A, B) - \mathrm{den}(X, Y)| \leq \epsilon$ for any two subsets $X \subseteq A$ and $Y \subseteq B$ satisfying $|X| \geq \epsilon|A|$ and $|Y| \geq \epsilon|B|$.

Definition 10. (ϵ-regular equipartition) A family of subsets $\mathcal{V} = \{V_1, \ldots, V_k\}$ ($V_i \subseteq V$, for all $i \in \{1, \ldots, k\}$) is called a *partition* of V if $V_i \cap V_j = \emptyset$ for all $1 \leq i < j \leq k$

and $V = V_1 \cup \cdots \cup V_k$. This k is called the *order* of the partition. A partition $\mathcal{V} = \{V_1, \ldots, V_k\}$ of the vertex set of a graph is called an *equipartition* if $|V_i|$ and $|V_j|$ differ by no more than 1 for all $1 \leq i < j \leq k$. An equipartition $\mathcal{V} = \{V_1, \ldots, V_k\}$ of the vertex set of a graph is called ϵ*-regular* if all but at most ϵk^2 of the pairs (V_i, V_j) $(i, j \in \{1, \ldots, k\})$ are ϵ-regular.

Now we can explain the lemma.

Theorem 11. [11, 104] (Szeméredi's regularity lemma) *For every pair of an integer t and a real number $\epsilon > 0$ there exists an integer $T = T_{11}(t, \epsilon)$ such that any graph with $n \geq T$ vertices has an ϵ-regular equipartition of order k, where $t \leq k \leq T$.*

Yao's minimax principle

In this subsection, we introduce Yao's minimax principle, which is based on the idea that any randomized algorithm can be regarded as a distribution over deterministic algorithms. We say that a deterministic algorithm A *errs in testing* a property $\mathcal{P}$ on an instance I if A rejects I if $I \in \mathcal{P}$ and A accepts I if I is ϵ-far from $\mathcal{P}$. We consider a distribution of instances $\mathcal{I}$. The subdistribution consisting of instances whose size is N is denoted by $\mathcal{I}_N$. Clearly $\mathcal{I} = \bigcup_{i=1}^{\infty} \mathcal{I}_i$.

Yao's minimax principle can be expressed in many different forms. The following is one in the property testing form.

Theorem 12. [21, 58, 107] (Yao's minimax principle) *Let $\mathcal{P}$ be a property and $q \colon \mathbb{Z}^+ \times \mathbb{R}^+ \to \mathbb{Z}^+$ be a function. Assume that for any $\epsilon > 0$ and for infinitely many $N \in \mathbb{Z}^+$, there exists a distribution $\mathcal{I}_N$ such that for every deterministic algorithm A whose query complexity is $q(N, \epsilon)$, the following holds:*

$$\Pr_{I \sim \mathcal{I}_N}[A \textit{ errs in testing } \mathcal{P} \textit{ on } I] > \frac{1}{3},$$

where $I \sim \mathcal{I}_N$ means that I is chosen according to distribution $\mathcal{I}_N$. Then the query complexity of any algorithm to test $\mathcal{P}$ on parameter ϵ and size N is more than $q(N, \epsilon)$.

Proof: Let $\mathcal{A}$ be an arbitrary randomized algorithm for testing $\mathcal{P}$ with query complexity at most $q(N, \epsilon)$. $\mathcal{A}$ is regarded as a distribution over deterministic algorithms. Thus the probability of $\mathcal{A}$ errs in testing $\mathcal{P}$ when instances are given over the distribution $\mathcal{I}_N$ is expressed as follows.

$$\Pr_{A \sim \mathcal{A},\ I \sim \mathcal{I}_N}[A \text{ errs in testing } \mathcal{P}] \geq \min_{A \in \mathrm{supp}(\mathcal{A})} \Pr_{I \sim \mathcal{I}_N}[A \text{ errs in testing } \mathcal{P}], \tag{3}$$

where supp($\mathcal{A}$) is the support of $\mathcal{A}$ which is the subset of non-zero values. From the assumption, (3) is greater than $1/3$. Therefore the probability of $\mathcal{A}$ errs in testing $\mathcal{P}$ when instances are given over the distribution $\mathcal{I}_N$ is more than 1/3. □

2.4 Basic Techniques

An elementary example of testers

Here we show an elementary (maybe naive) example of testers to help readers to understand how testers work. A function $f : \{0, \ldots, n\} \to \mathbb{R}$ is *linear* if there are real values $a, b \in \mathbb{R}$ such that $f(x) = ax + b$ for all $x \in \{0, \ldots, n\}$. [1] A function f is ϵ-far from being linear if for at least ϵn variables $x \in \{0, \ldots, n\}$, $f(x)$ must be changed to make f linear.

We will show a tester for testing linearity. The oracle of this problem, for any $x \in \{0, \ldots, n\}$, returns $f(x)$. In this algorithm we assume that $\epsilon \leq 1/4$. If $\epsilon \geq 1/4$, then using $\epsilon \leq 1/4$ is sufficient. The algorithm is the following.

procedure LINEARITY
begin
01 choose $s = 2\epsilon^{-1}$ values $S = \{x_1, x_2, \ldots, x_s\}$ from $\{0, \ldots, n\}$ independently and uniformly at random;
02 check whether all points $(x_i, f(x_i))$, $x_i \in S$ are collinear;
03 if they are collinear, then accept the input; otherwise, reject it;
end.

Theorem 13. LINEARITY *is a one-sided-error tester of linearity with query complexity* $O(\epsilon^{-1})$.

Proof: If the input is linear, then it is clearly accepted by the algorithm. Assume that the input is ϵ-far from being linear. Let B be one of the minimum sets of integers $i \in \{1, \ldots, n\}$ such that $(x_i, f(x_i))$ should be changed to make the input linear. From the assumption, $|B| > \epsilon n$. Thus the probability of a randomly chosen i being not in B is $1 - \epsilon$.

[1] In many articles of property testing, "linearity" is used in the different form. To give priority to good understanding of persons who are not familiar with this area, we use this definition.

Let L be the line formed by all the points that are not in $(x_i, f(x_i))(i \notin B)$. If S is collinear, then (i) $S \cap B = \emptyset$ or (ii) $|S - B| \leq 1$ (note that (ii) occurs only when the points in $S \cap B$ are accidentally on a line, say L_B, and note that L_B may cross L). The algorithm accepts the input by mistake in only these cases. The probability that (i) occurs is

$$(1-\epsilon)^s \leq (e^{-\epsilon})^s = e^{-\epsilon s} = e^{-\epsilon(2\epsilon^{-1})} = e^{-2} \leq \frac{1}{6}. \tag{4}$$

The first inequality is obtained from Lemma 7.

The probability that (ii) occurs is at most

$$s\epsilon^{s-1} = 2\epsilon^{s-2} = 2(1-(1-\epsilon))^{s-2} \leq 2\left(e^{1(1-\epsilon)}\right)^{s-2} = 2\left(e^{-1}\right)^{(1-\epsilon)(s-2)} \tag{5}$$

inequality is obtained from Lemma 7. From $1 - \epsilon \geq 3/4$ and $s = 2\epsilon^{-1} \geq 8$ (since $\epsilon \leq 1/4$),

$$(1-\epsilon)(s-2) \geq 3. \tag{6}$$

From (5) and (6), it follows that the probability that (ii) occurs is at most

$$2e^{-3} < \frac{1}{6}. \tag{7}$$

From (4) and (7), the probability that the algorithm accepts the input in mistake is at most $1/6 + 1/6 = 1/3$, i.e., the algorithm rejects it with probability at least $2/3$.

This algorithm is clearly one-sided-error, since every linear input is never rejected. The query complexity is $O(s) = O(\epsilon^{-1})$. □

Testing triangle-freeness on dense graphs

We show an example on graph-property testing. For an integer $n \in \mathbb{Z}^+$, we denote by K_n a complete n-graph. If a graph does not contain any K_3 as a subgraph, then it is said to be *triangle-free*. Triangle-freeness is clearly a property since any graph isomorphic to a triangle-free graph is triangle-free.

We first consider a tester for triangle-freeness in the dense-graph model, where the edge-oracle is used. We show a one-sided-error tester of triangle-freeness as follows, where $G = (V, E)$ is a given n-graph and s_ϵ is an integer that is fixed by ϵ and will be defined later.

procedure TRIANGLE-FREENESS

begin

01 choose $s = s_\epsilon$ vertices $S = \{v_1, v_2, \ldots, v_s\}$ from V independently and uniformly at random;

02 make the subgraph $G(S)$ induced by S through the oracle;

03 if $G(S)$ contains no K_3, then accept the input; otherwise, reject it;

end.

This algorithm is one-sided-error, since it rejects an input only if it finds a copy of K_3. Showing that it rejects every graph that is ϵ-far from being triangle-free with probability at least $2/3$ is not simple.

Lemma 14. [10] *For any $\epsilon > 0$, there is an integer $s = s_{14}(\epsilon)$ such that for any graph, if it is ϵ-far from being triangle-free, then a subgraph induced by s vertices chosen uniformly at random from the graph contains a K_3 with probability at least $2/3$.*

Proof sketch: Let $G = (V, E)$ be an n-vertex graph ϵ-far from being triangle-free. Let S be the set of vertices chosen by the above algorithm. From Theorem 11, it can be proven that there are $T = T_{14}(\epsilon)$, $\gamma = \gamma_{14}(\epsilon)$ and $t = t_{14}(\epsilon)$ that satisfy the following property: If $n \geq T$, then G has a γ-regular equipartition $\mathcal{V}$ of V with $t \leq |\mathcal{V}| \leq T$. Since G is ϵ-far from being triangle-free, (the detail is omitted but) it can be shown that there must be $W_1, W_2, W_3 \in \mathcal{V}$ such that $\mathrm{den}(W_i, W_j) \geq 2\gamma$ for all $1 \leq i < j \leq 3$. Since $\mathcal{V}$ is an equipartition, $|W_i|/n > 1/(2T)$ holds. From this, (the detail is omitted again but) it follows that if s is large enough, S includes three vertices $v_1 \in W_1$, $v_2 \in W_2$, and $v_3 \in W_3$ such that $(v_1, v_2), (v_2, v_3), (v_3, v_1) \in E$ with high probability. □

Theorem 15. [10] *Triangle-freeness on the dense-graph model is testable by* TRIANGLE-FREENESS *with one-sided error.*

Proof: We adopt $s_{14}(\epsilon)$ in Lemma 14 as s_ϵ in the procedure. If the graph is triangle-free, the algorithm clearly accepts it, and thus the algorithm is one-sided-error. Assume that the input graph is ϵ-far from being triangle-free. From Lemma 14, $G(S)$ contains at least one K_3 with probability at least $2/3$, and the input is rejected with probability at least $2/3$. □

Note that the query complexity of this algorithm is huge, since the constant $s_{14}(\epsilon)$ in Lemma 14 is a tower of ϵ^{-1}.

Parameter testing

We explain a method for approximating a value in constant time. Such a framework is sometimes called *parameter testing*.

Definition 16. *Let $x^* \in \mathbb{R}_0^+$ be a nonnegative real value. For a pair of nonnegative real values $\alpha \geq 1$ and $\beta \geq 0$, a value x is said to be an* (α, β)-approximation *of x^* if*

$$\frac{x^*}{\alpha} - \beta \leq x \leq \alpha x^* + \beta. \tag{8}$$

As an example, we show a constant-time $(1, \epsilon n)$-approximation algorithm for evaluating the number of edges of a given graph in the "bounded-degree-graph model." The *bounded-degree-graph model* (or the *bounded-degree model*, for short) only considers graphs such that every vertex has at most a constant number of neighbors, which is defined as follows.

Definition 17. (Degree and bounded-degree) We call the number of adjacent vertices of a vertex $v \in V$ of $G = (V, E)$ is the *degree* of v, which is denoted by $\deg_G(v)$, i.e., $\deg_G(v) := |\{w \in V \mid (v, w) \in E\}|$. The subscript G may be omitted if it is clear. For a positive integer $d \in \mathbb{Z}^+$, if $\deg_G(v) \leq d$ for every vertex $v \in V$ in graph $G = (V, E)$, then G is said to be *d-bounded-degree*. The set of d-bounded-degree graphs is denoted by $\Gamma(d)$. Sometimes d-bounded-degree is called *bounded-degree* for short.

The bounded-degree model considers only $\Gamma(d)$, where d is arbitrary. For any graph $G = (V, E) \in \Gamma(d)$, $|E| \leq dn/2$, where $n = |V|$, i.e., $|E| = O(n)$ for any constant d. Hence G is sparse. This means that the edge-oracle is useless, since for almost all queries, the answers are "There is no edge between the pair of vertices." Thus in the bounded-degree model, the following oracles are used.

- *Degree-oracle*: If an algorithm gives a vertex $v \in V$, this oracle returns $\deg(v)$.
- *Adjacent-vertex-oracle*: If an algorithm gives a pair of $v \in V$ and an integer $i \in \{1, \ldots, \deg(v)\}$, this oracle returns the ith vertex adjacent of v if exists, and 0 otherwise.

In this model, the denominator of the distance (Equation(1)) is dn.

Lemma 18. [65] *In the d-bounded-degree model, for any* $\epsilon > 0$ *and any* $0 < p < 1$, *a* $(1, \epsilon n)$*-approximation of* $|E|$ *can be obtained with probability at least* $1 - p$ *and query complexity* $O(d^2\epsilon^{-2} \log p^{-1})$.

Clearly $|E|$ can be expressed by the following equations, where Ex[deg] is the average degree of G:

$$|E| = \frac{1}{2} \sum_{v \in V} \deg(v) = \frac{n \cdot \mathrm{Ex}[\deg]}{2}. \tag{9}$$

By using this equation, we can construct an algorithm for estimating $|E|$ as follows.

procedure Graph-Size-Estimation
begin
01 choose $s = \frac{d^2}{8\epsilon^2} \ln \frac{2}{p}$ vertices $S = \{v_1, \ldots, v_s\}$ from V independently and uniformly at random;
02 calculate $\overline{\deg} = \frac{1}{s} \sum_{i=1}^{s} \deg(v_i)$ and $\overline{m} = n \cdot \overline{\deg}/2$;
03 **output** $\overline{m}$;
end.

Proof of Lemma 18: From Hoeffding's inequality (Theorem 8),

$$\Pr[|\overline{\deg} - \mathrm{Ex}[\deg]| \geq 2\epsilon] \leq 2 \exp\left(-\frac{2s^2(2\epsilon)^2}{sd^2}\right) = 2 \exp\left(\ln \frac{p}{2}\right) = p. \tag{10}$$

From $\overline{m} = n \cdot \overline{\deg}/2$ and $|E| = n \cdot \mathrm{Ex}[\deg]/2$, it follows that the above probability is equal to $\Pr[|\overline{m} - |E|| \geq \epsilon n]$. Therefore $\overline{m}$ is a $(1, \epsilon n)$-approximation of $|E|$. □

Lower bounds on query complexity

Some properties have been known to be non-testable. In this subsection we show how to prove non-testability by using examples.

A graph $G = (V, E)$ is called *bipartite* if V can be partitioned into two subsets V_1 and V_2 such that every edge is between V_1 and V_2, i.e., $E = E(V_1, V_2)$. Bipartiteness is clearly a property. Bipartiteness is the property that was first found to have a super-constant lower bound for testing. This was obtained by Goldreich and Ron [60].

Theorem 19. [60] *For the 3-bounded-degree (graph) model, there is a real number* $\epsilon > 0$ *such that any testing algorithm for bipartiteness with parameter* ϵ *requires* $\Omega(\sqrt{n})$ *queries, where n is the number of vertices.*

Proof sketch: In order to use Yao's minimax principle (Theorem 12), we construct a distribution $\mathcal{G}$ on 3-bounded-degree graphs such that any deterministic algorithm whose query complexity less than $\sqrt{n}/4$ cannot distinguish the given graph being bipartite from being 0.01-far from being bipartite with probability at least $2/3$.

$\mathcal{G}$ consists of two subsets $\mathcal{G}_1$ and $\mathcal{G}_2$: the former consists of bipartite graphs and the latter consists of graphs that are 0.01-far from being bipartite. A graph is given from $\mathcal{G}_1$ or $\mathcal{G}_2$ with the same probability, and any algorithm whose query complexity is small cannot distinguish a graph from $\mathcal{G}_1$ and a graph from $\mathcal{G}_2$ with high probability. We restrict that n to be even.

1. $\mathcal{G}_1$ consists of all 3-regular graphs that are composed of a Hamiltonian cycle and a perfect matching.
2. $\mathcal{G}_2$ consists of all 3-regular graphs that are composed of a Hamiltonian cycle and the perfect matching satisfying the following restriction: the distance on the cycle between every two vertices that are connected by a perfect matching edge must be odd.

Clearly all graphs in $\mathcal{G}_2$ are bipartite. It can be also proven that almost all graphs in $\mathcal{G}_1$ are 0.01-far from being bipartite.

Furthermore, it can be proven that any testing algorithm that performs $o(\sqrt{n})$ queries cannot distinguish between a graph chosen randomly from $\mathcal{G}_1$ and a graph chosen randomly from $\mathcal{G}_2$. □

In the same paper [60], it is shown that testing being an expander requires $\Omega(\sqrt{n})$ queries.

Stricter $\Omega(n)$ lower bounds on query complexity have been known for some properties. Bogdanov, Obata, and Trevisan [26] showed the first $\Omega(n)$ lower bound for Bounded-degree graph 3-colorability, and furthermore for Vertex Cover, Max Cut, Max 2SAT, Max E3SAT, [1] and Max E3LIN, [2] where all the problems above are on the

[1] EkSAT is SAT, with each clause has exactly k literals.

[2] EkLIN-h is the problem of deciding the satisfiability of a system of linear equations modulo h, with each equation has exactly k variables.

bounded-degree model, by introducing a reduction method. Later Yoshida and Ito [109] showed that 3-edge-colorability, Directed/undirected Hamiltonian path/cycle, 3-dimensional matching, and Schaefer-type generalized 3SAT, all in the bounded-degree model, have the same (linear) lower bounds by introducing some new reduction methods.

2.5 Characterizations of Testable Properties

One of the most attractive themes in the area of property testing to find a combinatorial characterization of testable properties.

Dense graphs

For the dense-graph model, a complete combinatorial characterization of testable property was found by Alon et al. [11]. To put it briefly, this characterization is expressed by the regularity lemma (in Subsection 2.3). To explain it a little more, the characterization is said to be "regular reducible," which is shown as follows.

Definition 20. (Regularity instance) A *regularity instance* R is given by an error-parameter $\epsilon > 0$, an integer k, a set of $\binom{k}{2}$ densities $0 \leq \eta_{i,j} \leq 1$ indexed by $1 \leq i < j \leq k$, and a set $\overline{R}$ of pairs (i, j) of size at most ϵk^2. A graph is said to satisfy the regularity instance if it has an equipartition $\{V_i \mid 1 \leq i \leq k\}$ such that for all $(i, j) \notin \overline{R}$ the pair (V_i, V_j) is ϵ-regular and satisfies $|E(V_i, V_j)| = \eta_{i,j}|V_i||V_j|$, i.e., $\mathrm{den}(V_i, V_j) = \eta_{i,j}$. The *complexity* of the regularity instance is $\max(k, 1/\epsilon)$.

Definition 21. (Regular-reducible) A graph property $\mathcal{P}$ is *regular-reducible* if for any $\delta > 0$ there exists $r = r_{\mathcal{P}}(\delta)$ such that for any n there is a family $\mathcal{R}$ of at most r regularity instances each of complexity at most r, such that the following holds for every $\epsilon > 0$ and every n-graph G.

1. If $G \in \mathcal{P}$, then for some $R \in \mathcal{R}$, G is δ-close to R.
2. If G is ϵ-far from $\mathcal{P}$, then for any $R \in \mathcal{R}$, G is $(\epsilon - \delta)$-far[1] from R.

Theorem 22. [11] *For the dense-graph model, a graph property is testable if and only if it is regular-reducible.*

For example, the triangle-freeness is regular-reducible.

Bounded-degree graphs

[1] Here we extend the term "ϵ-far" to negative ϵ, since for every instance can be regarded as ϵ-far (from any family of instances) for any $\epsilon \leq 0$ from the definition.

We have not obtained a complete characterization of testable properties in the bounded-degree (graph) model. However, an important sufficient condition called "hyperfiniteness" was found.

Definition 23. [67] Let $\epsilon > 0$, $t > 0$, and $d > 0$. Let $G = (V, E)$ be a d-degree-bounded n-graph. If one can remove at most ϵdn edges from G so that each connected component of the resulting graph has at most t vertices, then G is called (ϵ, t)-*hyperfinite* (with respect to degree bound d). For a function $\rho : \mathbb{R}^+ \to \mathbb{R}^+$, if G is $(\epsilon, \rho(\epsilon))$-hyperfinite for every $\epsilon > 0$, then G is called ρ-*hyperfinite*. A set $\mathcal{G}$ of d-degree-bounded graphs is called ρ-*hyperfinite* if every graph $G \in \mathcal{G}$ is ρ-hyperfinite. $\mathcal{G}$ is called *hyperfinite* if there is a function ρ such that $\mathcal{G}$ is ρ-hyperfinite.

Definition 24. [67] Let $\epsilon > 0, t > 0$, and $d > 0$. Let $G = (V, E)$ be an n-vertex d-degree-bounded graph. A partition $\mathcal{V}$ of V is called an (ϵ, t)-hyperfinite partition if $|W| \leq t$ for every $W \in V$ and the number of edges whose terminal vertices are in different sets of $\mathcal{V}$ is at most ϵdn.

Note that if a graph is hyperfinite, then it is close to a graph that can be partitioned into small connected components. If a graph can be partitioned into small connected components, then local search from randomly chosen vertices is well suited. In fact, the following theorem was given by Newman and Sohler [89].

Theorem 25. [89] *In the bounded-degree model, every graph property is testable for any hyperfinite family of graphs.*

Unfortunately this theorem is not a necessary condition, e.g., k-edge/vertex-connectedness for any fixed $k \geq 3$ is not hyperfinite but testable [60, 110]. However, a necessary condition based on hyperfiniteness was found by Fichtenberger, Peng, and Sohler et al. [49] as shown in the following.

A *subproperty* of a property $\mathcal{P}$ is a property that is a subset of $\mathcal{P}$. A property is *non-trivially testable* if it is testable and there exists $\epsilon > 0$ such that there is an infinite number of graphs that are ϵ-far from the property.

Theorem 26. [49] *Every testable property of bounded-degree graphs is either finite or contains an infinite hyperfinite subproperty. Also, the complement of every non-trivially testable graph property contains an infinite hyperfinite subproperty.*

Usually we suppose that testing algorithms know the size of the input, e.g., the number of vertices of a given graph. Alon and Shapira [12], however, introduced an

idea of *oblivious* tester, which must work without knowing the size, and showed a complete characterization of one-sider-error oblivious testers.

Recently, combinatorial characterizations of one-sided error testability for monotone and hereditary properties in the bounded-degree model were presented by Ito, Khoury, and Newman [70]. If for any $G \in \mathcal{P}$, a graph obtained by removing an arbitrary edge (resp., vertex) is in $\mathcal{P}$, then $\mathcal{P}$ is called *monotone* (resp., *hereditary*) [13]. Note that any minor-closed property [46] (including planarity) and k-colorability for any $k \in \mathbb{Z}^+$ (including bipartiteness) are monotone and hereditary. This characterization covers not only undirected graphs but also digraphs. In their paper, an idea of "forbidden configurations" was presented. This idea may be useful in obtaining a characterization of one-sided-error testable properties with no restriction on the bounded-degree model.

General graphs

We have shown two models on graphs: the dense-graph model and the bounded-degree model. Another popular model is the general (graph) model. This model is a generalization of the two models mentioned above.

In this model, an upper bound d of the *average* degree of every graph is given, i.e., $d \geq \sum_{i \in V} \deg(i)/n$ for every graph. Ordinarily we assume that d is a constant, and thus this model treats sparse graphs. The distance is defined as Equation (1), where d is the upper bound on the average degree. This model allows all oracles that can be used in the other two models.

As written above, this model is a generalization of the other two models and it is inevitably more difficult. Recently, models that focus on hierarchy have been researched. For example, there is research that focuses on graphs where the size distribution of the cliques shows a power law, and the degree distribution shows a power law after contracting those cliques [100]. This research is focused on the hierarchy of isolated cliques. A clique is a subgraph in which there exists an edge between every pair of vertices. For a nonnegative integer $c \geq 0$, a c-isolated clique is a clique, such that the number of outgoing edges (edges between the clique and the other vertices) is less than ck, where k is the number of vertices of the clique. A 1-isolated clique is sometimes simply called an *isolated clique* [68, 69]. There are other studies that focus on hierarchy, in a subclass of $\mathcal{SF}$ called $\mathcal{HSF}$, each of

which is defined as follows.

Definition 27. [66] Let $\mathcal{E}(G)$ be the graph obtained from G by contracting all isolated cliques. Two distinct isolated cliques never overlap, except in the special case of double-isolated-cliques, which consists of two isolated cliques with size k sharing $k-1$ vertices. A double-isolated-clique Q has no edge between Q and the other part of the graph (i.e., $dG(Q) = 0$), and thus we specially define that a double-isolated-clique in G is contracted into a vertex in $\mathcal{E}(G)$. Under this assumption, $\mathcal{E}(G)$ is uniquely defined.

Definition 28. [66] (Scale-Free (multi)graph) For positive real numbers $c > 0$ and $\gamma > 1$, a class of scale-free (multi)graphs $\mathcal{SF}(c, \gamma)$ consists of (multi)graphs $G = (V, E)$ for which the following condition holds. Let v_i be the number of vertices v with $\deg_G(v) = i$. Then,

$$v_i \leq cni^{-\gamma}, \text{for all } \gamma \in 2, 3, \ldots. \tag{11}$$

Definition 29. [66] (Hierarchical Scale-Free multigraphs; $\mathcal{HSF}$) For positive real numbers $c > 0, \gamma > 1$, and a positive integer $n_0 \geq 1$, a class of hierarchical scale-free (multi)graphs $\mathcal{HSF} = \mathcal{HSF}(c, \gamma, n_0)$ consists of (multi)graphs $G = (V, E)$ for which the following conditions hold.

(i) $G \in \mathcal{SF}(c, \gamma)$.

(ii) Consider the infinite sequence of graphs $G_0 = G, G_1 = \mathcal{E}(G_0), G_2 = \mathcal{E}(G_1), \ldots$. If $|V[G_i]| \geq n_0$, then G_i includes at least one isolated clique $Q \subseteq V$ with $|Q| \geq 2$. (Note that if G_k has no such isolated clique, then $G_k = G_{k+1} = G_{k+2} = \cdots$.)

In this $\mathcal{HSF}$, the following theorems are proven.

Theorem 30. [66] *For any $\mathcal{HSF} = \mathcal{HSF}(c, \gamma, n_0)$ with $\gamma > 2$ and any real number $\epsilon > 0$, there is a real number $t = t_{30}(\mathcal{HSF}, \epsilon)$, such that $\mathcal{HSF}$ is (ϵ, t)-hyperfinite.*

Theorem 31. [66] *Every property is testable for $\mathcal{HSF}(c, \gamma, n_0)$ with $\gamma > 2$.*

In the general-graph model, while no other universal (constant-time) tester has been known, universal testing algorithms with polylog(n)-time query complexity have been found on forests [80] and outerplanar graphs [15].

Other results

We briefly introduce some other results on not only characterizations but also various types of problems. On affine-invariant functions, Yoshida [108] presented a complete

characterization of testable properties.

Batu, Berenbring, and Sohler [19] gave a parameter-testing algorithm for the bin packing problem with query complexity $\tilde{O}(\sqrt{n} \cdot \text{poly}(\epsilon^{-1}))$. Ito, Kiyoshima, and Yoshida [71] showed a parameter-testing algorithm for the knapsack problem with query complexity $\tilde{O}(\epsilon^{-4})$. For EXPTIME-complete problems, the generalized chess, shogi (Japanese chess), and xiangqui (Chinese chess) were proven to be all testable by Ito, Nagao, and Park [72].

Lovász and Vesztergombi [82] introduced an idea of *nondeterministic* property testing, and showed a relation to deterministic (i.e., ordinary) property testing. In response to this paper Gishboliner and Shapira [56] an additional result using Szemerédi's regularity lemma.

2.6 Some Algorithms with Analogous Concepts

There are several algorithms that have similar concepts to sublinear progressive algorithms. In this subsection, we refer to the characteristics of these algorithms and their differences from progressive algorithms.

2.6.1 Online Algorithms

An *online algorithm* processes step by step, in the order that the input is given to the algorithm, i.e., it modifies the answer to fit each input when it is given. In contrast, an *offline algorithm* runs after reading all of the input [27]. For example, insertion sort is an online algorithm and selection sort is an offline algorithm. The online algorithm is defined as follows.

> *An online algorithm is one that receives a sequence of requests, and performs an immediate action in response to each request. Each sequence of requests and corresponding actions have associated cost. . . . The competitive ratio of an online algorithm is the maximum value of the ratio between the cost incurred by the online algorithm and the cost incurred by an optimal algorithm [74].*

In detail, they are defined as follows.

Definition 32. [74] (Online problems and algorithms) An *online problem* is specified by:

- A set R of *requests*;

- A set A of *actions*;
- A *cost function* $c\colon \bigcup_{n=1,2,\ldots} (R^n \times A^n) \to \mathcal{R}^+$ where $\mathcal{R}^+$ denotes the nonnegative reals.

For any request sequence $r \in R^n$, define $Opt(r)$ as $\min_{a \in A^n} c(r, a)$. An *online algorithm* $\mathcal{A}$ is determined by a function $f_{\mathcal{A}} : R^+ \to A$, where R^+ is the set of all finite nonempty sequences of requests. In response to a sequence of requests $r = r_1, r_2, \ldots, r_t$ the algorithm performs the sequence of actions $\mathcal{A}(r) = f_{\mathcal{A}}(r_1), f_{\mathcal{A}}(r_1 r_2), \ldots, f_{\mathcal{A}}(r_1 r_2 \ldots r_t)$ and incurs the cost $c(r, \mathcal{A}(r))$.

Definition 33. [74] (Competitive ratio) For any positive constant d, the online algorithm $\mathcal{A}$ is said to be *d-competitive* if there exists a constant b such that, for all request sequences r, $c(r, \mathcal{A}(r)) \leq dOpt(r) + b$. The *competitive ratio* of $\mathcal{A}$ is defined as the greatest lower bound of the set of c such that $\mathcal{A}$ is *c-competitive*.

Note that not all offline algorithms can be converted to online algorithms. Some algorithms can be converted depending on the problem, while others cannot. In other words, the online algorithm is an offline algorithm that allows sequential processing. It processes inputs that have arrived so far and does not know the future inputs, i.e., what and how many input will come. Certainly, its performance depends on the order in which the inputs are received. The online algorithm is designed to minimize the cost for the worst input. From this point of view, for evaluating the performance of an online algorithm, we use a measure called the competitive ratio, which compares with the cost of the optimal offline algorithm. It is important to design an online algorithm with a small competitive ratio and to prove upper or lower bounds of the competitive ratio. The online algorithm and sublinear progressive algorithms are similar in the idea of sequential processing. However, the sublinear progressive algorithm outputs a best-effort solution from the available input according to a pair of the approximation parameter and the fault probability. It is a solution that predicts and evaluates the entire input. In addition, there is a theoretical guarantee of the accuracy of the solution. When it is able to read all of the input, the sublinear progressive algorithm outputs the best solution from the input, which does not depend on the order of arrival.

2.6.2 Incremental Algorithm

Another algorithm with a similar concept is the incremental algorithm. Incremental algorithms, known as incremental computing and incremental learning, are used in

a variety of contexts and have different meanings. There are definitions by Giraud-Carrier [41] and by Zhou and Chen [111] in the context of machine learning, but these are not theoretical definitions. In this literature on machine learning, input is read from a dataset, but in incremental machine learning, input is given sequentially over time, and each time a new input is read, learning is performed, and the learning is updated according to the input. Amir et al. mention the difference between online algorithms and incremental algorithms in the context of machine learning.

> *We distinguish "online" from "incremental" learning. Online has to discard a sample after learning (no memory) and unlike to incremental learning is not allowed to store it [98].*

They define the difference between online and incremental as the difference in whether the input can be stored. In this field, they devise how to store the information of the input so far in a small number of areas. However, in this definition, the two algorithms are basically the same in the sense that they read the input from start to finish.

On the other hand, there is research that explicitly uses the name incremental in the theoretical field. For example, the following is a definition by Sharp.

> *An incremental algorithm is given a sequence of inputs, and finds a sequence of solutions that build incrementally while adapting to the changes in the input. . . . There are k time steps, or levels, defined by a sequence of k inputs. The goal is to produce a sequence of k outputs, one per time step, such that two constraints are satisfied. The first, known as the feasibility constraint, requires the level l output to be feasible with respect to the level l input. The second, known as the incremental constraint, requires the level l output to be a superset of that of level $l-1$, that is, the output builds incrementally. These two constraints define feasible incremental solutions; we can then evaluate and compare these solutions using a variety of objective functions, such as the aggregate value and competitive ratio objectives [99].*

In addition to this, she mentions the difference between online and incremental as follows.

> *Online algorithms also take in a sequence of inputs and produce incremental solutions; unlike their incremental counterparts, however, they do not know the input sequence in advance. We use our incremental results to better*

understand online algorithms and to indicate how their performance can be improved [99].

She defines an incremental algorithm as being able to know the input sequence in advance and incremental algorithm solutions to be inclusive.

Furthermore, there is another method, an algorithm called incremental computing [3, 4, 30, 50]. In this literature, incremental means that when data is updated, the output is changed according to the changed data.

Whereas incremental algorithms are used in a variety of contexts, the sublinear progressive algorithm is clearly different from them.

2.6.3 Anytime Algorithms

The definition of *Anytime algorithm* is as follows.

Definition 34. [23] (Anytime algorithms) *Anytime algorithms are defined as algorithms that return some answer for any allocation of computation time and are expected to return better answers when given more time.*

Anytime algorithms can be interrupted at any time to obtain an intermediate solution, and the accuracy of the intermediate solution improves with time. For example, heuristics such as simulated annealing and genetic algorithms are among the typical methods for anytime algorithms. Several notions have been proposed regarding the accuracy of intermediate solutions of anytime algorithms [112]. The anytime algorithm is intended to be a polynomial-time algorithm and it is similar to the idea of an incremental algorithm. Indeed, the sublinear progressive algorithm is a kind of anytime algorithm and it is close to the anytime algorithm with an accuracy guarantee, but there are three main differences.

- The sublinear progressive algorithm output solutions with an accuracy corresponding to the amount of input read.

  An anytime algorithm reads all of the input, keeps a tentative solution, and updates the accuracy of that solution, whereas the sublinear progressive algorithms read an input and improves the output solution over time.
- There are theoretical guarantees for execution time and error functions.

  The performance of anytime algorithms has been evaluated experimentally, whereas we show a formula for the relationship between computation time and accuracy for the sublinear progressive algorithms.

- Finally, a sublinear progressive algorithm outputs the exact solution.
  While anytime algorithms do not need to output the exact solution, the sublinear progressive algorithms eventually outputs the exact solution in the end.

2.6.4 Progressive Geometric Algorithms

In order to construct a framework for the sublinear progressive algorithm, we introduce the definition of the progressive geometric algorithm [8]. Alewijnse et al. do not only give two progressive geometric algorithms but also define a progressive geometric algorithm and explain the policy. The progressive geometric algorithm conbines a polynomial-time $(1 + \epsilon)$-approximation algorithm with an exact algorithm. In addition, they introduce ideal conditions that the solution error and computation time should satisfy. They define a progressive geometric algorithm as one that outputs partial solutions r times according to a *convergence function* that gives an upper bound on the error of a partial solution. They assume two ways of setting the running time.

- Set the maximum time to output each partial solution. The maximum time to wait for the next partial solution to be output can be set.
- Set the maximum amortized time T_r/r for the worst-case total running time T_r for outputting $1, \ldots, r$ partial solutions. The first solution is obtained early, and the running time increases as the later solutions are obtained.

Note that these settings are only a statement that such an optimization method exists. They do not suggest a general way to construct a progressive geometric algorithm, and there is no guarantee that a conversion that satisfies these two conditions exist. Their progressive geometric algorithm is structured to follow such a policy. In this progressive geometric algorithm, the convergence function only gives an upper bound on the error of partial solution, and then it is important to note that the error is not necessarily monotonically decreasing. Moreover, they write that monotonicity is desired as a property of a solution. This idea is similar to one of the incremental algorithms introduced in the previous section. The convex hulls algorithm presented in their paper is constructed so that the solution has monotonicity.

With that in mind, we would like to convert a sublinear-time algorithm into a progressive algorithm. Their method cannot be used directly because sublinear-time algorithms have two parameters: approximation parameter and fault probability. If

only the approximation parameter is sufficiently reduced with respect to the fault probability, we cannot say that the accuracy has improved, and vice versa. Both need to be decreased in a well-balanced. The solution must be output while taking into account the contribution of the approximation parameter and the fault probability to the computation time. In addition, it should be noted that the solution of sublinear algorithms (property testing) is not necessarily monotonic. If we let the solution be monotone, it is not inconceivable that we could store the sampled input and use it to search for the next solution. However, the randomness of the sampling is lost, and the effects of sampling bias cannot be ignored. Based on these properties and the framework of the sublinear-time algorithms, we need to construct the sublinear progressive algorithm. The two types of computation time settings they mention are both important. We ideally would like to perform an optimization that achieves both.

Chapter 3 Framework and Theorem of Sublinear Progressive Algorithms

3.1 Preliminaries

In this chapter, we present the idea of "sublinear progressive algorithms." Constant- or sublinear-time algorithms are useful when we need to get a solution quickly if a problem suddenly occurs and we should need to read huge data if we used a traditional polynomial-time algorithm. In such a case, it is useful to get an approximate solution quickly by using a constant- or sublinear-time algorithm. Even in such a case, however, after getting a temporary solution, we may hope to get a better solution gradually as time permits. Ideally, we would like to get better and better solutions gradually, and to get an exact solution finally at the same time as if we applied an exact algorithm from the beginning.

We call such algorithms "sublinear progressive algorithms[39]," which are illustrated in Fig. 2. Only preliminary results on this idea have been presented by the authors in an international conference[1] [38].

In this chapter, we show that for any problem, if there are both a constant-time algorithm and an exact algorithm, we can construct an ideal sublinear-time algorithm, i.e., given any positive integers $r \geq 2$ and $t \geq 1$, an algorithm that outputs r solutions

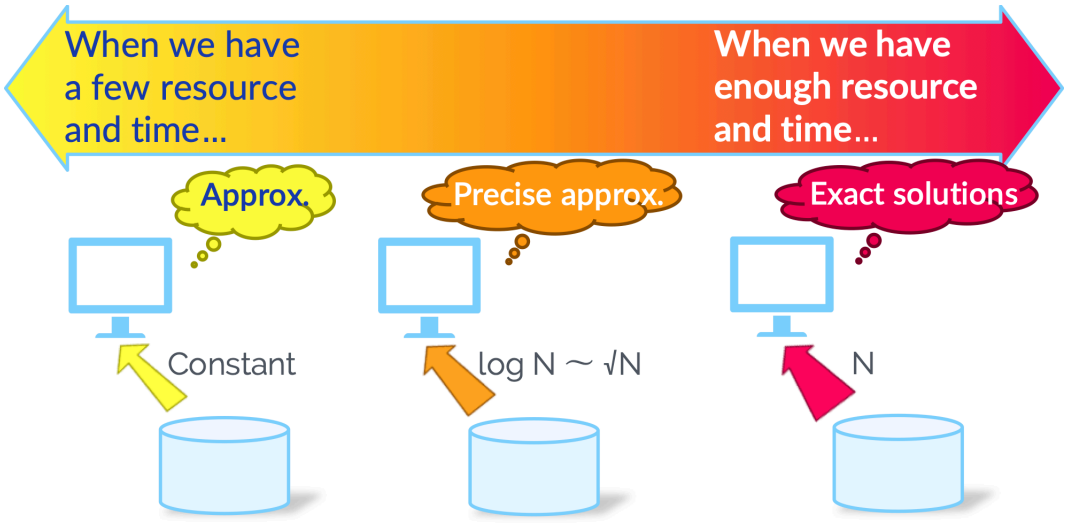


Figure 2: Sublinear Progressive Algorithms.

[1] Oral presentation only. No proceedings were published.

$S_1, S_2, \ldots, S_r$ in this order, where S_r is the exact solution which is obtained without error, under the constraint that

(1) S_1 is obtained in time at most t,

(2) for any $i \in \{1, \ldots, r\}$ the worst-case computation-time T_i for getting S_i is $O(T^*(S_i))$, where $T^*(S_i)$ is the computation time to get S_i by using the original constant-time algorithm (if $i \in \{1, \ldots, r-1\}$) or exact algorithm (if $i = r$), and

(3) $R := \max_{i \in \{2,\ldots,r\}} T_i/T_{i-1}$ is minimized in the big-O sense.

The meaning of Constraint (3) is that the time we should wait for the next solution follows a geometric progression. If it follows an arithmetic progression, T_2 is close to T_r/r and then T_r/r is close to T_r which is the time to output the final (exact) solution in the big-O sense. Thus, in this case, we must wait for a very long time (until T_r/r) to get the second solution if $T^*(S_r)$ is huge. By introducing the geometric progression, the waiting times become gradually longer and longer. It means that the time to obtain S_i is R times the total computation time to output S_{i-1}. It is to obtain progressively better solutions, which means incremental solution improvement.

3.2 Analysis of Fault Probability

In this subsection, we evaluate the fault probability of sublinear-time algorithms. A *Monte Carlo algorithm* that returns the correct answer with probability at least $\frac{1}{2} < p < 1$ is called a p-exact algorithm. When a p-exact algorithm outputs a yes-no answer, we can get a decision by majority vote. In other words, the algorithm is repeated multiple times to obtain a solution that accounts for more than half of the total. If p is greater than $\frac{1}{2}$, the probability that the majority decision is correct approaches 1 as the number of times the algorithm is repeated. We consider evaluating this probability as follows.

Lemma 35. (Success probability of majority voting) *For $\frac{1}{2} < p < 1$ ($p \in \mathbb{R}$), if the p-exact algorithm is run n times, the probability of obtaining the correct answer by majority vote is at least $1 - 1/e^{c_p n}$, where $c_p = \frac{1}{2p}(p - \frac{1}{2})^2$.*

To prove this lemma, we use the corollary obtained from Chernoff's inequality below.

Theorem 36. [36] (Chernoff's inequality) *Let $X_1, X_2, \ldots, X_n$ be mutually independent*

$\{0,1\}$ *random variables, and* μ *be the expected value of* $X := X_1 + \cdots + X_n$. *For any* $\lambda > 0$,

$$Pr[X \geq (1+\lambda)\mu] < \left(\frac{e^{\lambda}}{(1+\lambda)^{(1+\lambda)}}\right)^{\mu},$$
$$Pr[X \leq (1-\lambda)\mu] < \left(\frac{e^{\lambda}}{(1-\lambda)^{(1-\lambda)}}\right)^{\mu}.$$

Corollary 37. [87] *Let* $X_1, X_2, \ldots, X_n$ *be mutually independent* $\{0,1\}$ *random variables, and* μ *be the expected value of* $X := X_1 + \cdots + X_n$. *For any* $0 < \lambda \leq 1$,

$$Pr[X \geq (1+\lambda)\mu] \leq \exp\left(-\frac{\lambda^2\mu}{3}\right)$$
$$Pr[X \leq (1-\lambda)\mu] \leq \exp\left(-\frac{\lambda^2\mu}{2}\right)$$

Proof of Lemma 35: Let the p-exact algorithm run n times, and the probability of getting the correct answer by majority vote is at least

$$\sum_{j=\lfloor n/2 \rfloor + 1}^{n} \binom{n}{j} p^j (1-p)^{n-j},$$

where j is the number of successes. Let X be a random variable representing the number of correct answers,

$$Pr[X = k] = \binom{n}{k} p^k (1-p)^{n-k},$$

which is clearly a binomial distribution. From Chernoff's inequality, we can estimate the probability of getting the correct answer by majority voting. From Corollary 37, calculate the probability that the majority vote fails, that is, the probability that X is less than the majority. Since the expectation of the binomial distribution is $\mu = np$, if $\lambda = 1 - \frac{1}{2p}$, then $(1-\lambda)\mu = \frac{n}{2}$, and the following holds:

$$Pr[X < n/2] = Pr[X < (1-\lambda)\mu] \leq e^{-\frac{1}{2p}n(p-\frac{1}{2})^2} = \frac{1}{e^{c_p n}}, \tag{12}$$

where $c_p = \frac{1}{2p}(p-\frac{1}{2})^2$ (c_p is a constant determined by the probability p). Therefore, the probability of getting the correct answer by majority vote is at least $1 - 1/e^{c_p n}$. □

Corollary 38. *Let* q *be the fault probability of a majority vote if a algorithm with fault probability* p *is run* k *times. The following inequality is satisfied:*

$$k \geq \frac{1}{c_p} \log_e \frac{1}{q},$$

where $c_p = \frac{1}{2p}(p-\frac{1}{2})^2$.

Proof of Corollary 38: From Equation (12) of Lemma 35, $1/e^{c_p k}$ is less than or equal to q,

$$\begin{aligned}
\frac{1}{e^{c_p k}} \leq q \iff & \quad \tfrac{1}{q} \leq e^{c_p k} \\
\iff & \quad \log_e \tfrac{1}{q} \leq \log_e e^{c_p k} \\
\iff & \quad \log_e \tfrac{1}{q} \leq c_p k \\
\iff & \quad \tfrac{1}{c_p} \log_e \tfrac{1}{q} \leq k \quad \square
\end{aligned}$$

3.3 SPA Theorem

We present *Sublinear Progressive Algorithm Theorem* (*SPA Theorem*, for short) as follows.

Definition 39. *Let $\mathcal{P}$ be a property. For positive real numbers $\epsilon > 0$ and $0 < p < 1/2$, a random valuable $S \in \{yes, no\}$ which is the output of testing algorithm for $\mathcal{P}$ is said to be an $(\epsilon, 1-p)$-solution. If the input is in $\mathcal{P}$, S = yes with probability at least $1-p$. If the input is ϵ-far from $\mathcal{P}$, S = no with probability at least $1-p$.*

Our algorithm, given $r \geq 2$, outputs r solutions $S_1, S_2, \ldots, S_r$, where S_i is an $(\epsilon_i, 1-p_i)$-solution for $i = 1, \ldots, r-1$, and S_r is the exact solution. Let T_i be the time to get S_i by the progressive algorithm. Condition (2) in Subsection 3.1 means that every intermediate solutions and the final solution must be calculated in almost the same time as the original constant-time or exact algorithms. In order to satisfy Condition (3), these solutions are obtained in time following a geometric progression, i.e., $T_i = O(t\tau^{i-1})$ for some $\tau > 1$. Moreover, from the requirement of actual applications, ϵ_i and p_i should decrease gradually.

The following theorem shows that for any property, if it has a constant-time algorithm and an exact algorithm (an exponential-time algorithm is allowed), then a sublinear progressive algorithm satisfying the above conditions exists.

Theorem 40. (SPA Theorem: property testing version) *Let $\mathcal{P}$ be a property. Suppose that there exist a tester* Alg_0 *whose time complexity is $T^*(\epsilon, 1-p)$, where $\epsilon > 0$ is an approximation parameter and $0 < p \leq 1/2$ is a fault probability, and an exact algorithm* Alg_1 *whose time complexity is $T^*(n)$, where n is the size of the input, for $\mathcal{P}$. Then there exists an algorithm, given any positive integers $r \geq 1$ and $0 < t \leq T^*(n)$, that provides r solutions S_1, S_2, ..., S_r, in time T_1, T_2, ..., T_r, respectively, and*

satisfies the following conditions:

(1) $T_1 = \Theta(t)$.

(2.1) S_r *is the exact solution and* S_i *is an* $(\epsilon_i, 1 - p_i)$*-solution for some* $\epsilon_i > 0$ *and* $0 < p_i \leq 1/2$*, for all* $i \in \{1, \ldots, r-1\}$.

(2.2) $T_r = O(T^*(n))$ *and* $T_i = O(T^*(\epsilon_i, 1 - p_i))$*, for all* $i \in \{1, \ldots, r-1\}$.

(3) For $i \in \{1, \ldots, r-1\}$, $T_i = O(t\tau^{i-1})$ *for there exists* $\tau \geq 2$ *(*τ *is independent of* i*), and moreover if* $\tau > 2$*, then* $T_i = \Theta(t\tau^{i-1})$.

(4) Both ϵ_i *and* p_i *decrease with increasing* i.

Note that in $O(*)$ and $\Theta(*)$ in this theorem, ϵ, r, and t are regarded as variables. Although we did not define ϵ_r and p_r in this theorem, we can introduce them as $\epsilon_r = 0$ and $p_r = 0$ (since S_r is the exact solution).

In Condition (3) of this theorem, the common ratio of increasing T_i is at least 2, i.e., $\tau \geq 2$. This is because if τ is smaller than 2, then we do not need to calculate solutions so frequently, and it is sufficient to use 2 as the common ratio. Moreover, the latter half of Condition (3) "if $\tau > 2$, then $T_i = \Theta(t\tau^{i-1})$" is important. If this constraint does not exist, $T_i = O(t\tau^{i-1})$ (for $i \in \{1, \ldots, r-1\}$) can be trivially satisfied by letting τ be very huge, e.g., $\tau \geq T^*(n)/T_1$.

Proof of Theorem 40: All of Conditions (*) appearing in this proof represent the conditions in the statement of this theorem.

From Condition (1), we calculate appropriate ϵ_1 and p_1 that satisfy $T^*(\epsilon_1, 1-p_1) \leq \max\{2t, t + c\}$ for an appropriate constant $c > 0$, and then the algorithm is required to output $S_1 \in \{\text{yes}, \text{no}\}$ when t is small. Note that since any random solution is an $(\epsilon, 1/2)$-solution for any ϵ, there must be such ϵ_1 and p_1. Hence,

$$T_1 := \max\{2t, t + c\}. \tag{13}$$

Let τ and $T'_1, \ldots, T'_r$ be as follows, where n is the size of the input.

$$\tau = \max\left\{\left(\frac{T^*(n)}{T_1}\right)^{\frac{1}{r-1}}, 2\right\}, \tag{14}$$

$$T'_i := \min\{T_1\tau^{i-1}, T^*(n)\} \quad (i = 1, \ldots, r). \tag{15}$$

If $(T^*(n)/T_1)^{1/(r-1)} < 2$ in (14), then $\tau = 2$ and T'_i may reach $T^*(n)$ before $i = r$, and the algorithm can stop there, i.e., if there exists k such that $T'_{k-1}\tau \geq T^*(n)$, then

we stop the increase of T'_i at this point, i.e., $T'_k = T'_{k+1} = \cdots = T'_r$. Let k be the minimum positive integer that satisfies the above condition. Note that $k < r$ occurs only if $\tau = 2$. From these definitions, $T'_1 = T_1$ and $T'_k = T'_r = T^*(n)$.

For each $i = 2, \ldots, k-1$, we calculate ϵ_i and p_i that satisfy $T^*(\epsilon_i, 1-p_i) = \Theta(T'_i)$. Since $T^*(\epsilon, 1-p)$ is a decreasing function of both ϵ and p and $T^*(\epsilon, 1-p) \leq T^*(n)$ for any $0 \leq \epsilon \leq 1$ and $0 \leq p \leq 1$ (because Alg_1 outputs a $(0,1)$-solution and thus if $T^*(\epsilon, 1-p) > T^*(n)$ for some ϵ and p, then we can replace Alg_0 with Alg_1 for such ϵ and p), there must be such ϵ_i and p_i satisfying $\epsilon_i \leq \epsilon_{i-1}$ and $p_i \leq p_{i-1}$.

Now we present the progressive algorithm. Note that $k < r$ occurs only if $\tau = 2$, and otherwise $k = r$.

procedure PROGRESSIVE
begin
01 **do from** $i = 1$ **to** $k-1$;
02 **if** $T'_i = T^*(n)$ **then stop**;
03 calculate ϵ_i and p_i;
04 **call** $\mathrm{Alg}_0(\epsilon_i, 1-p_i)$ and **output** the solution (S_i);
05 **enddo**
06 **call** Alg_1 and **output** the solution (S^*);
end.

We show that this algorithm satisfies the conditions. Conditions (1), (2.1) and (4) are clear from the construction of the algorithm. From $T'_1 = T_1 = \max\{2t, t+c\}$, Condition (2.2) is clear for $i = 1$.

For every $i \in \{2, \ldots, k\}$,

$$\begin{aligned} T_i &= \sum_{j=1}^{i} T'_i = \sum_{j=1}^{i} T'_1 \tau^{j-1} = \frac{1}{\tau - 1}(\tau^{i-1} - 1)T'_1 \\ &\leq 2\tau^{i-1} T'_1 \quad (\text{because } \tau \geq 2) \\ &= 2T'_i. \end{aligned}$$

From this, it follows that $T_k = T_r = O(T^*(n))$ and $O(T^*(\epsilon_i, 1-p_i))$ for $i \in \{2, \ldots, k-1\}$, i.e., Condition (2.2) is satisfied for all i. The former half of Condition (3) is clear.

If $\tau > 2$, then $k = r$ and the latter half of Condition (3) is satisfied. □

Theorem 40 assures that we can construct a progressive algorithm without losing any computation time in the big-O sense. Furthermore, it can be observed that the above algorithm is the optimum in some meaning as shown below.

Observation 41. *The algorithm constructed in the proof of Theorem 40 is an algorithm that minimizes the ratio* $R := \max_{i\in\{2,\ldots,r\}} T_i/T_{i-1}$ *in the big-O sense, among all algorithms that satisfy all of the conditions.*

Proof of Observation 41: Let $\widehat{T}_i$ be the time when the i-th solution S_i is output by a constant-time algorithm. To obtain the exact solution S_r we need at least $T^*(n)$ time, i.e., $\widehat{T}_r \geq T^*(n)$. Moreover, the first solution S_1 must be outputted by sublinear progressive algorithm in time $\Theta(t)$. Since $T_1 = \Theta(t)$, we have $\widehat{T}_1 \leq c'T_1$ for some constant $c' \geq 1$. Thus,

$$\frac{\widehat{T}_r}{\widehat{T}_1} \geq \frac{T^*(n)}{c'T_1} = \frac{\tau^{r-1}}{c'} \quad \text{(by (14) and } \tau \geq 2).$$

From

$$\widehat{T}_r \leq R^{r-1}\widehat{T}_1,$$

it follows that

$$\begin{aligned} R &\geq \left(\frac{\widehat{T}_r}{\widehat{T}_1}\right)^{\frac{1}{r-1}} \geq \left(\frac{1}{c'}\right)^{\frac{1}{r-1}} \tau \geq \frac{\tau}{c'} \quad \text{(because } c' \geq 1) \\ &= \Omega(\tau). \end{aligned}$$

□

3.4 Representative Example of ϵ_i and p_i

We consider a representative case on sublinear progressive algorithms. From Corollary 38, in many constant-time algorithms, the computation time for getting an $(\epsilon, 1-p)$-solution can be represented by

$$t(\epsilon, 1-p) = c' \left(\frac{1}{\epsilon}\right)^{c''} \log_2 \frac{1}{p}, \tag{16}$$

where c' and c'' are constants that depend on the property.

Under this assumption we show the concrete expressions of ϵ_i and p_i that appeared in Theorem 40. However, there is a freedom to fix the ratio between ϵ_i and p_i, i.e., even if the value of $t(\epsilon_i, 1 - p_i)$ is fixed, there are two valuables, ϵ_i and p_i, and we cannot fix them.

In the proof of Theorem 40, the computation time of S_i is longer than that of S_{i-1} at the ratio of τ, i.e., $t(\epsilon_i, 1 - p_i)/t(\epsilon_{i-1}, 1 - p_{i-1}) = \tau$. This τ can be divided into the ϵ-dependent ratio, $(1/\epsilon)^{c''}$, and the p-dependent ratio, $\log_2(1/p)$. Here, we separate computation time as $\rho : 1 - \rho$ for arbitrary $0 \leq \rho \leq 1$ in every step, i.e., the ϵ-dependent ratio becomes τ^ρ times longer and the p-dependent ratio becomes $\tau^{1-\rho}$ times longer in each step.

The freedom in fixing ϵ_1 and p_1 still remains. If p_1 is fixed, then from $t(\epsilon_1, 1-p_1) = T_1$ and (16), we obtain

$$\epsilon_1 = \left(\frac{c' \log_2 \frac{1}{p_1}}{T_1} \right)^{\frac{1}{c''}}. \tag{17}$$

Note that $\epsilon_i \leq 1$. From this, $c' \log_2(1/p_1) \leq T_1$, and thus

$$p_1 \geq \frac{1}{2}^{T_1/c'}. \tag{18}$$

By setting c in (13) to be larger than or equal to this c', p_1 can be at most $1/2$. We can choose any p_1 that satisfies (18). From the above discussions, ϵ_i and p_i for $i \in \{2, \ldots, r-1\}$ are expressed as follows.

$$\epsilon_i = \frac{\epsilon_1}{\tau^{\rho(i-1)/c}}, \tag{19}$$

$$p_i = p_1^{\tau(1-\rho)(i-1)}. \tag{20}$$

3.5 Considerations and Observations on Application

Accuracy

Note that the accuracy in sublinear progressive algorithms is a pair of approximation parameter and fault probability. A property testing algorithm outputs yes or no as the answer. Even though this answer is a yes-no answer, it has a fault probability. In other words, even if the algorithm outputs yes, it does not necessarily mean that

the true answer is yes. For example, we run the sublinear-time algorithm 10 times with (ϵ_1, p_1) and got all yes. Next, we run the sublinear-time algorithm 20 times with (ϵ_2, p_2) and got 18 times yes and 2 times no. The percentage of yes has dropped from 100% to 90%. This kind of output can be obtained despite the correct conditions such that $\epsilon_1 \geq \epsilon_2$ and $p_1 \geq p_2$. At first glance, this appears to violate the condition that the accuracy of the solution is monotonically decreasing. Under these situations, we consider the results of ϵ_2 and p_2 to be more accurate. The reason is that the algorithm often outputs yes all 10 times by chance. The decrease of the yes proportion does not decrease the accuracy. The pair of ϵ_i and p_i itself is interpreted as accuracy.

Why to run the exact algorithm only once at the end?

If you give a sublinear-time algorithm $(\epsilon, p) = (0, 0)$ as input, it output the same solution as the exact solution in principle. Although it can be regarded as an exact algorithm, the sublinear progressive algorithm runs the exact algorithm only once at the end. The sublinear-time algorithm with $(\epsilon, p) = (0, 0)$ has some overhead compared to the exact algorithm because the sublinear-time algorithm originally outputs the answer from a part of the input. This means that before ϵ and p reach 0, the computation time of the sublinear-time algorithm is the same as the computation time of the exact algorithm. If it is able to read all the input, then it is appropriate to switch to the exact algorithm.

There is another reason for such a framework. The sublinear progressive algorithm is a combination of the constant-time algorithm and the exact algorithm for convenience. Here, the exact algorithm means an algorithm that reads all inputs, so the exact algorithm can be replaced by any algorithm that reads all inputs. In other words, for example, our method allows for the combination of any sublinear-time and polynomial-time approximation algorithms. The combination of the polynomial-time approximation algorithm and the exponential-time exact algorithm is as described in related work. By combining our method with previous research, we can smoothly switch between algorithms.

Chapter 4 K_3 edge cover problem in a wide sense

4.1 Introduction

In this chapter, we consider combinatorial optimization problems that have important applications in the design of experiments.

Problem 1 (reshown)**.** There are several samples and machines. Each sample can only be evaluated relatively. The number of samples that a machine can hold is fixed. You have to compare several combinations several times. If the number of samples is (a), the number of machines is (b), the number of samples the machine can hold is (c), and the number of comparisons is (d), what combination minimizes the number of comparisons?

This problem is closely related to the block design and schoolgirl problems [78].

- If $a = 15, b = 5, c = 3, d = 1$, then it is a schoolgirl problem.
- If $b = \frac{(a-1)}{2}, c = 3, d = 1$, it is a generalized schoolgirl problem.
- If a, b, c, and d are all variables, it is the block design.

This problem can be formulated as a problem in graph theory. We cover all edges of a given complete graph $G = (V, E)$, $|V| = a$, with b cliques of size c. The number of times an edge should be covered is d. What do the given graph and an edge mean in the design of experiments? It represents a combination to be compared, and a complete graph indicates that pairs must be compared. If an input graph is not a complete graph, some pair with no edges represents no need for comparison. This is a natural extension and has already been studied as a problem of edge covering [93]. It is NP-hard in the case where c is a variable and the problem of minimizing b for any a and $d = 1$.

We discussed the case of comparing if there is an edge or not comparing if there is no edge. In many cases of actual experiments, it is likely that there are pairs that do not need to be compared, and a pair can be compared more than once. We consider a further extension of this problem. What would be the extension if there are some intermediate states between covered or uncovered? It corresponds, in graph theory, to covering the edges of the original graph by allowing "spilling-out" and overlap on the edges. This is a very simple setting, but there is no result of edge coverings that consider "spilling-out." We call this problem the *K_k edge cover problem in a*

wide sense [37]. The problem can be formulated as follows. For a given graph $G = (V, E)$, find the minimum number of k-cliques (K_ks) that cover all edges of G. Multiple covering or covering one edge by more than one k-cliques is allowed. "Spilling-out" means that a set of k vertices can be covered by a k-clique even if the induced subgraph by them is not a clique. The K_2 edge cover problem in a wide sense is trivial because the minimum number of cliques is the number of edges in the graph. In this chapter, we prove that the case $k = 3$ is NP-hard.

4.2 Definitions

We consider only simple undirected graphs (graphs with no self-loop and no parallel edge). The number of edges of graph G is denoted as $||G||$. We call a subgraph (or its vertex set) that is a complete graph *clique*. A clique consisting of k vertices is called a *k-clique* and denoted by K_k. For a graph $G = (V, E)$, a family $\mathcal{X} = \{X_1, \ldots, X_p\}$ of vertex subsets is called a *K_k edge cover in a wide sense* (or a *K_k edge cover* for short) if (1) $|X_i| = k$ for any $i \in \{1, \ldots, p\}$ and (2) for any edge $(u, v) \in E$, there is a vertex subset $X_i \in \mathcal{X}$ such that $u, v \in X_i$. In this case, we say that edge (u, v) is *covered* by X_i. We call p the size of the K_k edge cover. The minimum size of a K_k edge cover is denoted by $\gamma_k(G)$. In the K_k edge cover problem, if one edge e is contained in a clique in a solution set, then we say e is covered. In this dissertation, we consider only the case of $k = 3$, i.e., the K_3 edge cover problem in a wide sense. The problem is defined as follows.

Problem K_3-EDGE-COVER-IN-A-WIDE-SENSE (K_3EC)

Instance: A graph $G = (V, E)$, and a positive integer $h \geq 1$.

Question: $\gamma_3(G) \leq h$?

For this problem we obtain the following results. Let C_4 and C_5 be the cycles of length 4 and 5, respectively.

Theorem 2. *K_3EC is NP-complete even if graphs are restricted to planar, cubic, and C_4, C_5-free as subgraphs (i.e., not restricted to induced ones).*

Theorem 3. *For K_3EC, there is an $O(|E||V| + 2^k|E|)$-time algorithm, where k is the number of 3-cliques in G.*

Theorem 4. *For K_3EC, if a tree-decomposition of tree-width t is given, there is an $O(2^{2(t+1)(t+2)}t^2|V|)$-time algorithm.*

4.3 Related Work

The school girl problem, block design, and covering by cliques have been extensively studied. The school girl problem is a classical combinatorial mathematical problem [78], and fast algorithms for solving this problem have been studied [18]. Block design is a well-known problem in the field of discrete mathematics [106], concrete patterns [75] and applications [92] have been considered. For more details on block design and graphs, see [32], [45], and [83].

There are related studies on the problem of covering a vertex by paths [63, 88]. K_3EC is directed related to the following problems. Problem Partition-into-cliques, PIC for short, is a problem, given a graph $G = (V, E)$ and an integer K, for deciding whether or not V can be partitioned into $k(\leq K)$ cliques. Problem Covering by cliques, CBC for short, is a problem, given a graph $G = (V, E)$ and an integer K, for deciding whether or not V can be covered by $k(\leq K)$ cliques which are subgraphs of G and cover (include) all edges in E. PIC can be regarded as a problem of covering vertices by cliques. CBC can be regarded as a problem covering edges by cliques with edge repetitions allowed. These two problems are known to be NP-complete [73, 93]. Regarding PIC, polynomial-time algorithms are known for circular arc graphs [55], for chordal graphs [54], for comparability graphs [61]. Regarding CBC, intersection number is the smallest number of cliques which cover (include) all edges in E [73], and an FPT algorithm with parameter k (the size of the solution) is given [43, 62]. K_3EC differs from them in the following two parts: (1) it allows spilling-out and (2) the size of the clique is limited. There have been no studies on this problem. Clearly, the problems of covering by K_1 or K_2 are trivial. For the problem of covering by P_2, which is a path consisting of two edges, a polynomial-time algorithm is easily obtained by modifying our algorithm given in the proof of Lemma 45. If there is no size restriction of cliques, K_3 edge cover in wide sense becomes trivial, since covering by a $|V|$-clique is optimal.

4.4 NP-Completeness

In this section, we show a proof of Theorem 2.

Theorem 2 (reshown)**.** *K_3EC is NP-complete even if graphs are restricted to planar, cubic, and C_4, C_5-free as subgraphs* (*i.e., not restricted to induced ones*)*.*

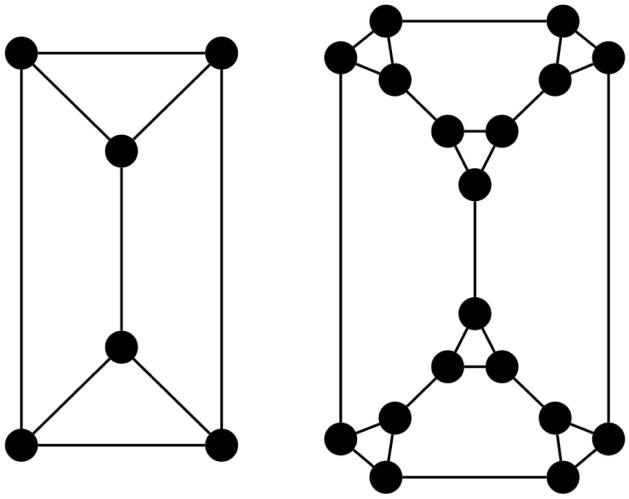


Figure 3: Reduction to K_3EC from IS.

K_3EC is clearly in NP. Thus we will show NP-hardness. It will be done by reducing the *Maximum Independent Set (IS)*, which is a well-known NP-complete problem [53].

Let $H = (W, F)$ be a graph. A vertex subset $V' \subseteq V$ such that there is no edge between any vertices of V' is called an *independent set*, and $|V'|$ is its *size*. The problem is defined as follows.

Problem Maximum Independent Set (IS)

Instance: A graph $H = (W, F)$, and a positive integer $h \geq 1$.

Question: Does H have an independent set with size at least h?

Theorem 40. [52] *IS is NP-complete even if H is planar and cubic.*

We show how to reduce IS to K_3EC. Let $(H = (W, F), h)$ be an instance of IS, where H is cubic and planar. We construct an instance $(G = (V, E), k)$ of K_3EC as follows.

Intuitively, we obtain G from H by replacing a vertex $w \in W$ with a K_3 with vertices w_0, w_1, and w_2; next, connecting the three edges e_0, e_1, and e_2 incident to w (note that H is cubic) to w_0, w_1, and w_2 one by one. See an example of this reduction in Fig. 3. Note that if H is a cubic planar graph, then G is cubic, planar, and C_4, C_5-free. Let $k = \frac{5}{2}n - h$, where $n = |W|$.

It is clear that this reduction can be done in polynomial-time. Thus it is sufficient to show that H has an independent set of size h if and only if G has a K_3 edge cover

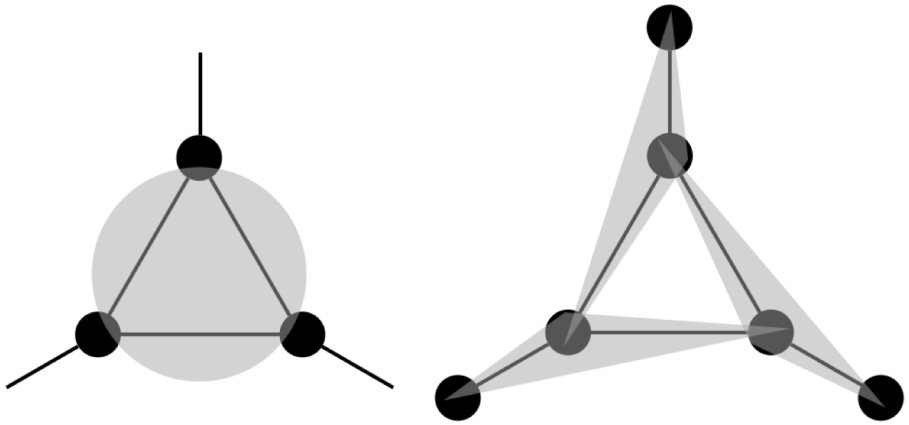


Figure 4: Delta-triangle (left) and $3L$-triangle (right).

of size $k = \frac{5}{2}n - h$.

Before showing it, we introduce some terms. Each K_3 of G, which corresponds to a vertex in W, is called a *triangle*. The three edges in a triangle are called *triangle-edges*. On the other hand, edges connecting distinct triangles are called *link-edges*. If two distinct triangles are connected by a link-edge, then they are called *adjacent*.

Let $\mathcal{X}$ be a K_3 edge cover of G of size p, where p is an integer. If $X \in \mathcal{X}$ covers three edges, then X is called a *delta*. If $X \in \mathcal{X}$ covers just two edges, then X is called an L. If $X \in \mathcal{X}$ covers only one edge, then X is called an I. If a triangle is covered by a delta, then the triangle is called a *delta-triangle*. If a triangle is covered by three Ls, then the triangle is called a *3L-triangle* (see Fig. 4).

If all triangles are covered by delta-triangles or $3L$-triangles, the K_3 edge cover is called *regular*. If there is no pair of adjacent $3L$-triangles in a regular K_3 edge cover, the K_3 edge cover is called *independent regular*.

Lemma 41. *If H has an independent set $U \subseteq W$ with $|U| = h$, then G has a K_3 edge cover $\mathcal{X}$ with $|\mathcal{X}| = \frac{5}{2}n - h$.*

Proof: We construct $\mathcal{X}$ from U as follows. For a triangle in G, if the corresponding vertex in H is in U, then we let the triangle be covered by a $3L$, and otherwise be covered by a delta. The remaining edges in G are covered by Is. See Fig. 5 for example.

We calculate the size of $\mathcal{X}$. Let $n = |W|$. Since $H = (W, F)$ is cubic, $|F| = \frac{3}{2}n$. From this, it follows that the number of triangles and link-edges in G are n and $\frac{3}{2}n$,

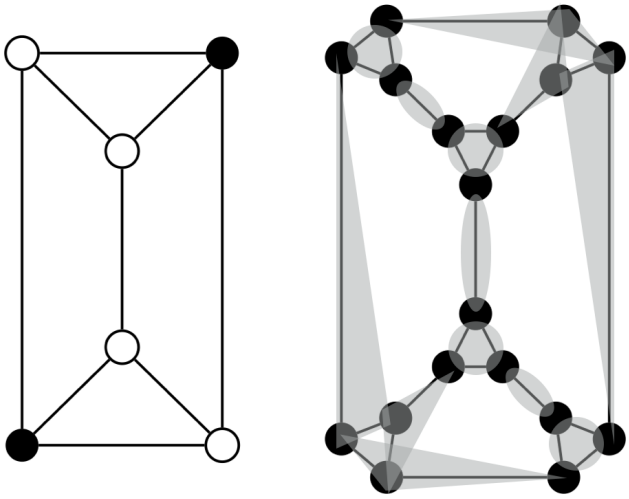


Figure 5: U (left) and the corresponding $\mathcal{X}$ (right): In the left figure, black vertices are in U, the triangle covered by a $3L$ in $\mathcal{X}$.

respectively. Since the number of $3L$-triangles is h, the number of delta-triangles is $n - h$. The number of link-edges covered by $3L$ is $3h$, and the number of link-edges covered by Is is $\frac{3}{2}n - 3h$. By summing up the numbers of deltas, $3Ls$, and Is used in $\mathcal{X}$, we get the following equations:

$$|\mathcal{X}| = (n - h) + 3h + \frac{3}{2}n - 3h = \frac{5}{2}n - h. \tag{21}$$

Therefore, $\mathcal{X}$ is the desired K_3 edge cover. □

Note that the K_3 edge cover $\mathcal{X}$ obtained above is independent regular. To show the reverse of this lemma is a little more complicated. We first show the following lemma.

Lemma 42. *If G has a K_3 edge cover $\mathcal{X}$, then G has an independent regular K_3 edge cover $\mathcal{X}'$ with $|\mathcal{X}'| \leq |\mathcal{X}|$.*

Proof: First we construct a regular K_3 edge cover $\mathcal{X}'$ with $|\mathcal{X}'| \leq |\mathcal{X}|$ from $\mathcal{X}$. At the first step, let $\mathcal{X}'$ be equal to $\mathcal{X}$. If $\mathcal{X}'$ is not regular, we modify $\mathcal{X}'$ to be regular by using the following operations.

- **Operation I:** If there is a pair $X, X' \in \mathcal{X}'$ such that all edges covered by X' are also covered by X, then remove X' from $\mathcal{X}'$.
- **Operation II:** If there is $X \in \mathcal{X}'$ that covers only one triangle-edge and no link-edge, then X is replaced with a delta covering the triangle-edge together

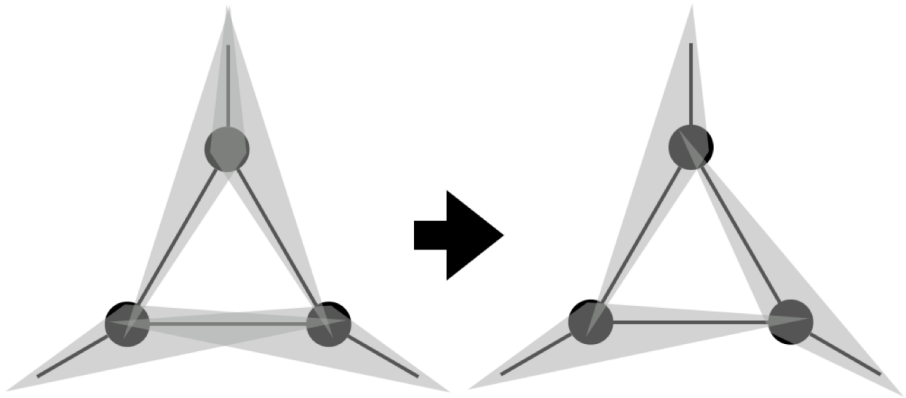


Figure 6: Operation Ⅳ.

with the other two triangle-edges in the triangle.

- **Operation Ⅲ:** If a triangle-edge e is covered by a delta and an L, then the L is replaced with an I covering only e', where e' is the other edge covered by L.
- **Operation Ⅳ:** If the three edges of a triangle are covered by more than three Ls, these Ls are changed to a $3L$ (at least one L is removed; see Fig. 6).

An algorithm for changing $\mathcal{X}'$ to be regular is the following.

procedure REGULARIZE($\mathcal{X}'$)
  begin
    do while $\mathcal{X}'$ is not regular;
      if Operation I can be applied **then** apply Operation I;
      else if Operation Ⅱ can be applied **then** apply Operation Ⅱ;
      else if Operation Ⅲ can be applied **then** apply Operation Ⅲ;
      else if Operation Ⅳ can be applied **then** apply Operation Ⅳ;
      end if
    enddo
  end
end procedure

We show that the above procedure stops in finite steps for any input $\mathcal{X}'$. Each operation never increases the size of $\mathcal{X}'$. Moreover, both the number of Ls and the number of $X \in \mathcal{X}'$ that cover only one triangle-edge never increase too. Operations I and Ⅳ decrease the size of $\mathcal{X}'$, and thus they can be applied finite times. Operations Ⅱ and Ⅲ decrease the number of $X \in \mathcal{X}'$ that cover only one triangle-edge and the

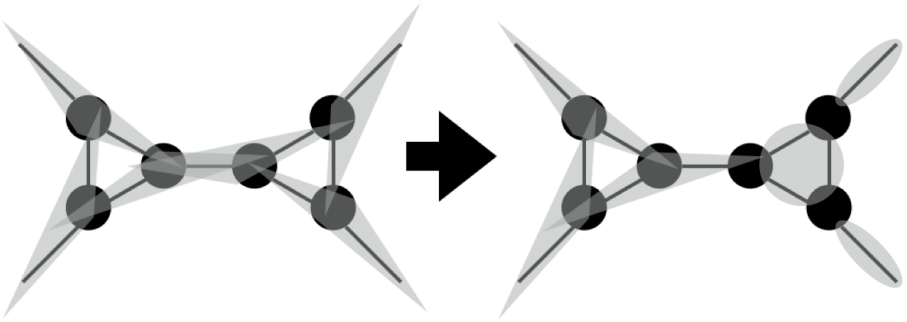


Figure 7: Operation V.

number of Ls, respectively, and thus they can be applied finite times. From these discussions, Procedure Regularize stops in finite steps, totally. It is clear that if Regularize($\mathcal{X}'$) stops, $\mathcal{X}'$ is regular, because if $\mathcal{X}'$ is not regular, then operations I-V are applicable.

Now we obtain a regular K_3 edge cover $\mathcal{X}'$. We next change it to an independent regular K_3 edge cover.

- **Operation V:** If there is a pair of adjacent $3L$-triangles, then replace one of the two Ls which share the same edge (e.g., the central edge of Fig. 7) with a delta-triangle, and apply Operation Ⅲ to the triangle (see Fig. 7).

Operation V does not increase the size of the cover. By applying the above operation whenever possible, $\mathcal{X}'$ finally becomes independent regular. □

Now we show the reverse of Lemma 41.

Lemma 43. *If G has a K_3 edge cover $\mathcal{X}$ with $|\mathcal{X}| = \frac{5}{2}n - h$, then H has an independent set $U \subseteq W$ with $|U| = h$.*

Proof: Assume that G has a K_3 edge cover $\mathcal{X}$ with $|X| = \frac{5}{2}n - h$. From Lemma 42, G has an independent regular K_3 edge cover $\mathcal{X}'$ with $|\mathcal{X}'| \leq |\mathcal{X}|$. Since there are no adjacent $3L$-triangles in G,

$$U := \{w \in W \mid \text{the corresponding triangle in } G \text{ is a } 3L\text{-triangle in } \mathcal{X}'\} \tag{22}$$

becomes an independent set. From the discussion made in the proof of Lemma 41, $|\mathcal{X}'| = \frac{5}{2}n - |U|$. Thus if $\frac{5}{2}n - |U| = |\mathcal{X}'| \leq |\mathcal{X}| = \frac{5}{2}n - h$, then $|U| \geq h$. Therefore any $U' \subseteq U$ with $|U'| = h$ is the desired independent set of H. □

Now we establish the proof of Theorem 2.

Proof of Theorem 2: Follows directly from Lemmas 41 and 43. □

Chapter 5 FPT algorithms

As shown by Theorem 2, K_3EC is NP-hard even for planar and cubic graphs. In this chapter, we consider two FPT algorithms. Not all NP-hard problems have FPT algorithms. In addition, note that the computational complexity varies depending on the problem. For example, there is an $O(2^k n^2)$ algorithm for Vertex Cover, but there is no $O(n^{o(k)})$ algorithm for Independent Set under the assumption of FPT $\neq W[1]$. Each NP-hard combinatorial optimization problem requires a specialized FPT because the problem settings are different. Keeping the above in mind, we consider two FPT algorithms.

The first one is FPT algorithm with the number of K_3s as a parameter in Section 5.1. In graph theory, the *girth* is the length of a shortest cycle contained in a graph [46]. If the graph does not contain any cycle, its girth is defined to be infinity [101]. A graph with girth 4 or more is triangle-free. In a triangle-free graph, K_3 can cover at most two edges. It is important to know how many K_3s are included in a graph. We considered FPT algorithm with this parameter.

The second is an FPT algorithm with the tree-width as a parameter in Section 5.2. If a tree-decomposition is a mapping of a graph to a tree, and a tree-width is a parameter that represents the "tree-ness" of the graph. If we can convert a given graph to a tree decomposition, the algorithm can treat the graph like a tree. This means that the obtained tree can be traversed exactly once bottom-up, which means that dynamic programming on the tree decomposition is easier to construct. Unfortunately, the problem of finding the tree-width t has been shown to be NP-complete [14]. However, for graphs with small tree-width t, polynomial time algorithms for input n are known. Bodlaender proved the existence of the $O(t^{O(t^3)} \cdot n)$-time algorithm [24]. This is still being improved, and there have been many studies of computation time for various approximation [25, 51]. In addition, there is a nice tree decomposition, which has good properties that make it easy to consider dynamic programming. Klok showed that a tree decomposition of tree-width t can be converted to a nice tree decomposition of tree width t in $O(tn)$-time [79]. Hence, we consider dynamic programming on nice tree decomposition and FPT with tree-width. There are few studies on covering with width as a parameter, but there are some researches on vertex cover [16, 105].

5.1 A Parameter of the Number of K_3

In this section, we prove the following theorem.

Theorem 3 (reshown)**.** *For K_3EC, there is an $O(mn + 2^k m)$-time algorithm, where k is the number of 3-cliques in G.*

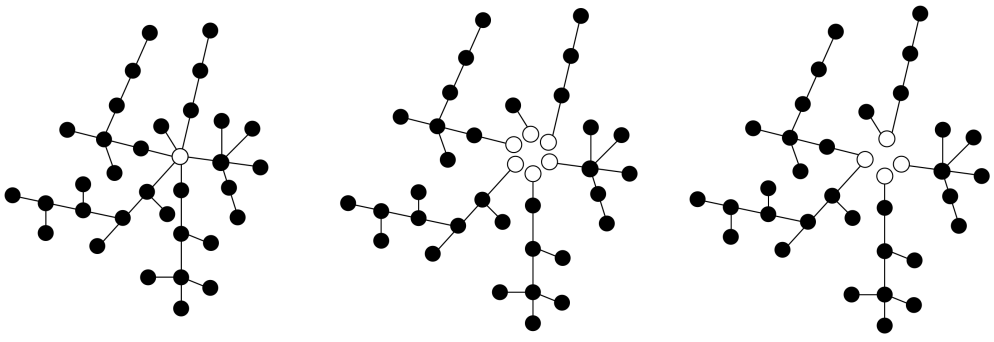
5.1.1 K_3 Edge Cover Problem in a Wide Sense on Trees

First, we consider K_3EC in a tree, prove the following lemma.

Lemma 44. *If $T = (V, E)$ is a tree, $\gamma_3(T) = \lceil |E|/2 \rceil$.*

Proof: If T is a path, $\gamma_3(T) = \lceil ||T||/2 \rceil$ is trivial. We consider the case where there is a vertex with degree 3 or more. We call such a vertex a *branching vertex* (see Fig. 8 (a)).

Let v be a branching vertex. We divide T into a set of subtrees $\{T_1, \ldots, T_k\}$ (where k is the degree of v) according to the following rule: each T_i is a maximal subtree such that v is one of its leaves (see Fig. 8 (b)). The *size* of a subtree T_i is defined as $||T_i||$. If the size is odd, the subtree is called an *odd-subtree* and otherwise it is called an *even-subtree*. If there are two odd-subtrees, we join them into one even-subtree. By applying this as far as possible, we finally get a set of subtrees $\{T'_1, \ldots, T'_k\}$ which includes at most one odd-subtree (see Fig. 8 (c)).



(a) (b) (c)

Figure 8: (a) example of tree (one of branch point is the white circle), (b) branch point partition $\{T_1, \ldots, T_6\}$, (c) $||T_i||$ is connected to $\{T_1', \ldots, T_4'\}$.

By applying this operation to each subtree recursively, we finally get a set of paths which consists of at most one path with odd size. For each path P, $\gamma_3(P) = \lceil ||P||/2 \rceil$.

By summing them up, we obtain the following upper bound:

$$\gamma_3(T) \leq \lceil ||T||/2 \rceil. \tag{23}$$

Next, we show the lower bound. Since K_3 is not included in the tree, each X can cover at most two edges in any K_3 edge cover $\{X_1, \ldots, X_p\}$, and thus the following inequality holds:

$$\gamma_3(T) \geq \lceil ||T||/2 \rceil. \tag{24}$$

From Eqs. (23) and (24), $\gamma_3(T) = \lceil |E|/2 \rceil$ is obtained. □

5.1.2 P_2 Edge Cover Problem in a Wide Sense

Covering with P_2 (a path of length 2) corresponds to that K_3EC without using deltas. Hence, we consider the following lemma.

Lemma 45. *For any connected graph $G = (V, E)$, the minimum size of K_3 edge cover problem in a wide sense without using deltas is $\lceil |E|/2 \rceil$.*

To prove this lemma, we use the following operation. Let $G = (V, E)$ be a graph and $v \in V$ be a vertex with degree at least two. Delete v together with the edges incident to v from G and add two new vertices v' and v''. New edges are added as follows. The set of vertices adjacent to v is denoted by $W = \{w \in V \mid (v, w) \in E|\}$. Let X and Y be two non-empty partition of W such that $X \cup Y = W$, $X \cap Y = \emptyset$, and $|X|, |Y| \neq \emptyset$, and add edge sets $E_X = \{(v', w) \mid w \in X\}$ and $E_Y = \{(v'', w) \mid w \in Y\}$ to E. The above operation is denoted by a *vertex-division* on v, i.e., it is defined by getting $G' = (V', E')$ such that $V' = (V - \{v\}) \cup \{v', v''\}$ and $E' = (E - \{(v, w) \in E \mid w \in W\}) \cup E_X \cup E_Y$. Note that since there is arbitrariness in the way of dividing W into X and Y, the result of a vertex partition is not defined uniquely if the degree of v is more than two.

Proof of Lemma 45: By applying the vertex-division on an arbitrary vertex v on an arbitrary cycle of $G = (V, E)$, the resulting graph is still connected and the number of edges does not change, and the number of cycles decreases at least by one. Thus by applying a finite number of vertex-divisions to G, we get a tree T that has the same number of edges as G. From Lemma 44, $\gamma_3(T) = \lceil ||T||/2 \rceil = \lceil |E|/2 \rceil$. Since T is the spanning tree of G, and for any pair of adjacent edges in T, the corresponding pair of edges are also adjacent in G, there is a K_3 edge cover in G with the size of $\gamma_3(T)$. On

the other side, the size of K_3 edge cover in G without deltas is clearly at least $\lceil |E|/2 \rceil$. From the above discussion, this lemma is obtained. □

5.1.3 The Number of 3-cliques in the Graph as a Parameter

We prove Theorem 3 using Lemma 45 as follows.

Proof of Theorem 3: We give an algorithm for solving the problem. The algorithm focuses on an arbitrary triangle of G. It divides the case into two cases according to whether or not the triangle is covered by a delta. If it is covered by a delta, the algorithm deletes the three edges in the triangle from G. Otherwise, the algorithm gives a label "not covered by a delta" to the triangle. The algorithm recursively applies the above operations as far as an unlabeled triangle exists. The graph finally obtained is not allowed to be covered by using any delta, and the solution can be obtained from Lemma 45. From that, the computation time of each case is $O(m)$ and the number of branches is at most k and the algorithm requires $\Theta(mn)$ time for enumeration all triangles [40]. Therefore, the total computation time is $O(mn + 2^k m)$. □

5.2 Tree-width as a Parameter

Here we give an FPT algorithm with width as a parameter.

Theorem 4 (reshown)**.** *For K_3EC, if a tree-decomposition of width t is given, there is an $O(2^{2(t+1)(t+2)}t^2 n)$-time algorithm.*

First, we assume that all graphs considered here are connected; otherwise, it is enough to manage each connected component one by one. [1] Under this assumption, we do not need to consider I (K_3 covering only one edge) as shown in the following.

Lemma 46. *If $G = (V, E)$ is connected and $|V| \geq 3$, there is an optimal solution that includes no I.*

Proof: Since G is connected and $|V| \geq 3$, every edge has at least one adjacent edge. Hence if an edge is covered by an I, we can replace this I with a delta or an L (K_3 covering two edges). The size of K_3 edge covers does not change by this change. By applying this operation whenever an I exists, we finally obtain a K_3 edge cover that uses no I with the same size. □

Next, we introduce some terms and prepare some other lemmas.

[1] This is valid for K_3 edge covers. If we use K_4, this strategy does not necessarily work.

Definition 47. [79] (Tree-decomposition) *A tree-decomposition of a graph* $G = (V, E)$ *is a pair* $D = (S, T)$ *with a family* $S = \{B_1, \ldots, B_r\}$ *of subsets of* V *and a tree* $T = (I, F)$ *such that* $I = \{1, \ldots, r\}$ *and the following three conditions are satisfied. For every* $i \in I$, i *is called a node and* B_i *is called a bag.*

(1) $\bigcup_{i \in I} B_i = V$,

(2) for every edge $(v, w) \in E$, *there is at least one bag* B_i *such that* $(v, w) \in B_i$, *and*

(3) for each vertex v *the set of nodes* $\{i \in I \mid v \in B_i\}$ *forms a subtree of* T.

Definition 48. [79] (Tree-width) *The width of a tree-decomposition* $D = (S, T)$ *is defined as* $\max_{i \in I}(|B_i| - 1)$. *The tree-width of a graph* G *is the minimum possible width through all tree-decompositions of* G.

Definition 49. [79] (Nice tree-decomposition) *A tree-decomposition* $D = (S, T)$ *is called nice if the following four conditions are satisfied.*

(1) T *is a rooted tree and each node has at most two children.*

(2) If a node i *has two children* j *and* k, *then* $B_i = B_j = B_k$.

(3) If node i *has a unique child* j, *then "*$|B_i| = |B_j| + 1$ *and* $B_j \subset B_i$ *" or "*$|B_i| = |B_j| - 1$ *and* $B_i \subset B_j$*".*

(4) Every bag corresponding to a leaf node of D *consists of only one vertex.*

Definition 50. [79] (Node types of a nice tree-decomposition) *In a nice tree-decomposition* $(\{B_i \mid i \in I\}, T = (I, F))$ *every node has one of the following four possible types.*

Leaf: A node that is a leaf.

Introduce: A node i *that has a child* j *and* $|B_i| > |B_j|$.

Forget: A node i *that has a child* j *and* $|B_i| < |B_j|$.

Join: A node that has two children.

Lemma 51. [79] *If a tree-decomposition with width* t *of graph* G *is given, a nice tree-decomposition of width* t *having* $O(|V(G)|)$ *nodes can be obtained in* $O(t^2 \cdot \max(|V(T)|, |V(G))|))$*-time.*

5.2.1 Preliminaries

Our algorithms use a nice tree-decomposition obtained by using Lemma 51. From here every tree-decomposition appearing below is a nice tree-decomposition, unless otherwise stated. In what follows a tree decomposition is regarded as a nice tree decomposition. Let i and j be nodes in I and let B_i and B_j be bags corresponding to i and j, respectively. If j is a descendant (an ancestor, resp.) of i, then B_j is called a

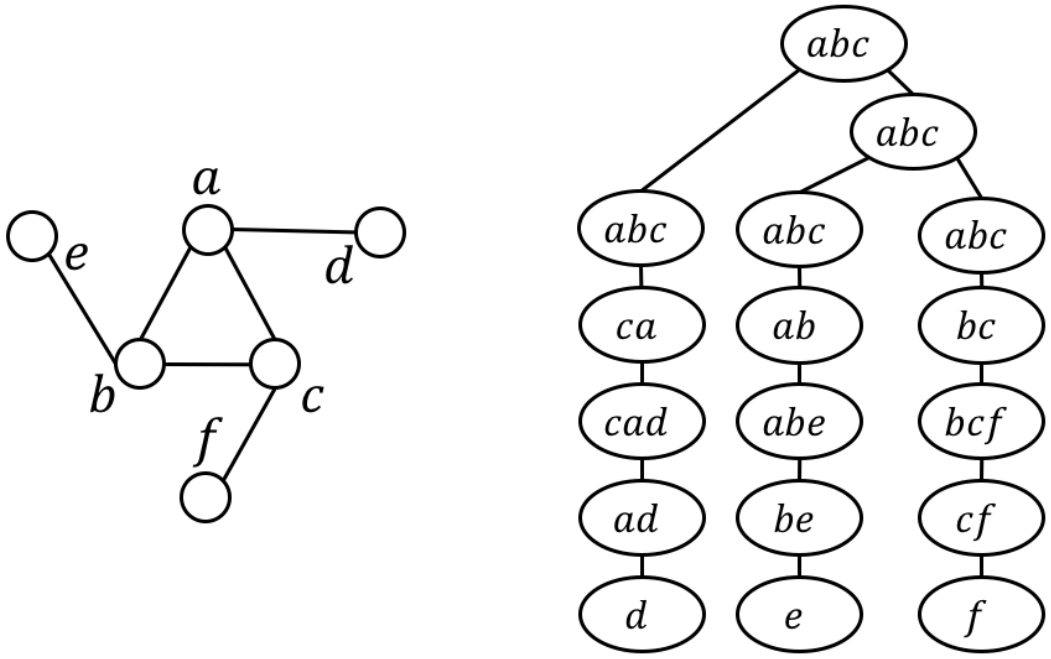


Figure 9: An example of a graph and a nice tree-decomposition.

descendant (an ancestor, resp.) of B_i. The *lower vertex set* $B_\downarrow$ of a bag B is defined as follows:

$$B_\downarrow := \{v \in B' \mid B' \text{ is a descendant of } B\} - B. \tag{25}$$

Let $B^+_\downarrow := B_\downarrow \cup B$. The upper vertex set $B_\uparrow$ of the bag B is $B_\uparrow = V - B^+_\downarrow$. Also, $B^+_\uparrow := B_\uparrow \cup B$. For an arbitrary K_3 edge cover $\mathcal{X} = \{X_1, \ldots, X_p\}$ of a graph G and an arbitrary vertex subset $W \subseteq V$, let $\mathcal{X}_W := \{X_i \in \mathcal{X} \mid X_i \subseteq W\}$, which is a partial solution of $\mathcal{X}$ for W. The edge set of the subgraph of $G = (V, E)$ induced by $W \subseteq V$ is denoted by E_W.

Let $G_{B^+_\downarrow} = (B^+_\downarrow, E_{B^+_\downarrow})$ be the subgraph of $G = (V, E)$ induced by $B^+_\downarrow$. Let $E^-_{B^+_\downarrow, \mathcal{X}}$ be the set of edges not covered by $\mathcal{X}_{B^+_\downarrow}$ in $E_{B^+_\downarrow}$. $\mathcal{X}^*_W$ is called an optimal partial solution for W if $\mathcal{X}^*$ is the minimum size of the K_3 edge cover problem.

Our algorithm traces the rooted tree T in the postorder, and for each bag B, constructs a set of partial solutions for $B^+_\downarrow$ such that one of them is an optimal partial solution. For saving the memory, we compress the data of the partial solutions and store them as a table. We will explain the details in the following.

For an intuitive explanation, let us assume a bag B has k vertices. For the bag $B = \{v_1, v_2, \ldots, v_k\}$, we create a table $T[B]$, which is a compressed representation of

the partial solutions. Let $G_B := (B, E_B)$ be the subgraph of G induced by B. Let $E_B^+ := \{(v, w) \in E_{B_\downarrow^+} \mid \{v, w\} \cap B \neq \emptyset\}$, i.e., E_B^+ is the set of edges in $E_{B_\downarrow^+}$ that are incident to at least one vertex in B. Let E'_B be $E'_B := E_B^+ - E_B$, i.e., the set of edges in E_B^+ that are incident to just one vertex in B. Let $h = |E_B|$ be the number of edges between vertices of B, and we give serial numbers to these edges as $e_1, \ldots, e_h$.

Let $\mathcal{X}^j$, $j = \{1, \ldots, q\}$, be the partial solutions stored in $T[B]$. $\mathcal{X}^j$ is represented as row j of $T[B]$. Each row of $T[B]$ is divided into three parts: the first part consists of $k(= |B|)$ columns, the second part consists of h columns, and the third part is only one column (see Fig. 10 for an example of $T[B]$. Note that in this figure, the tree-decomposition is not nice so as to make it simple). That is, $T[B]$ consists of $k + h + 1$ columns. The i-th cell of the first part of row j stores the number of edges between v_i and vertices in $B_\downarrow$ and uncovered by $\mathcal{X}^j$ (Note that the set of these edges is $E^-_{B_\downarrow^+, \mathcal{X}^j} \cap E'_B$). It will be shown later (in Lemma 54) that this number is enough to be 1 or 0. The i-th cell of the second part of row j is 1 if e_i is covered by $\mathcal{X}^j$; otherwise 0. The size $|\mathcal{X}^j|$ of $\mathcal{X}^j$ is stored in the third part of row j. For notational simplicity, this value $|\mathcal{X}^j|$ is represented by ω in the following. The compressed expression of a partial solution excluding the third part is called the *signature* of the partial solution.

In the following, we show some lemmas necessary for supporting our algorithm.

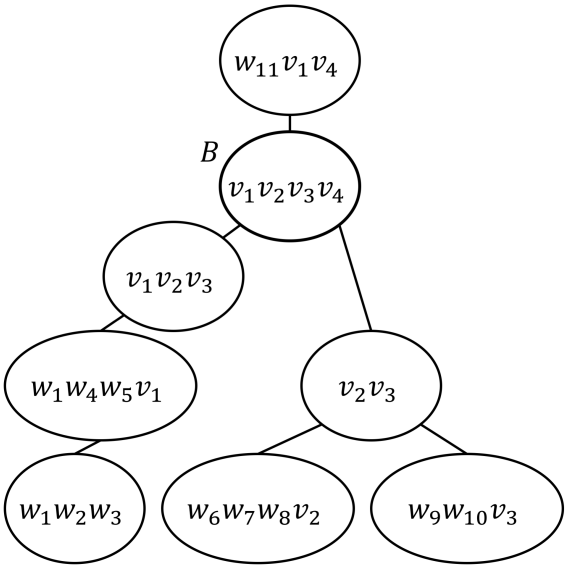
Lemma 52. *For any bag B of a tree-decomposition of a graph, and two vertices $u \in B_\uparrow$ and $v \in B_\downarrow$, there is no edge between u and v, i.e., $(u, v) \notin E$.*

Proof: From Condition (3) of Definition 47, there is no bag containing both u and v, hence $(u, v) \notin E$. □

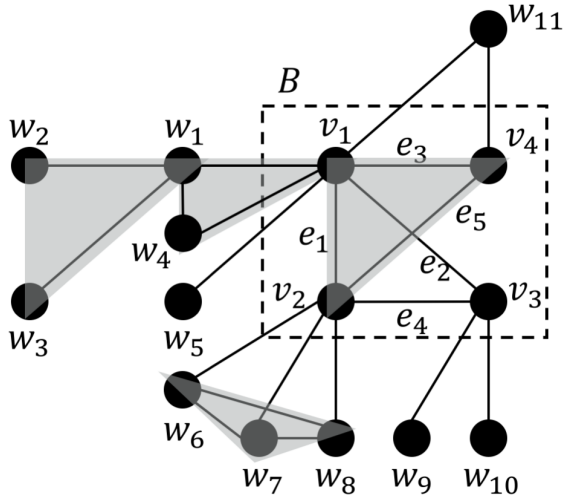
Lemma 53. *For any optimal solution $\mathcal{X}$ and a bag B, all edges belonging to $E^-_{B_\downarrow^+, \mathcal{X}}$ are incident to vertices in B.*

Proof: Assume that an edge $(u, v) \in E^-_{B_\downarrow^+, \mathcal{X}}$ is not incident to any vertex in B, i.e., $u, v \notin B$. Since $E^-_{B_\downarrow^+, \mathcal{X}} \subseteq E_{B_\downarrow^+}$, it follows that $u, v \in B_\downarrow$. Let X be an element of $\mathcal{X}$ that covers (u, v). Since $X \notin \mathcal{X}_{B_\downarrow^+}$ and $(u, v) \in E^-_{B_\downarrow^+, \mathcal{X}}$, X covers at least one edge besides edge (u, v). [1] Let the edge be (v, w) without loss of generality. Here, since

[1] Since X is an element of $\mathcal{X}$ and $X \notin \mathcal{X}_{B_\downarrow^+}$, it means that one of the vertices in X is not in $B_\downarrow^+$. X should contain another vertex w other than u and v that is not in B, because $u, v \in B_\downarrow \subseteq B_\downarrow^+$. Therefore, we have that X covers at least one edge besides (u, v), say, (u, w) or (v, w) (I does not exist from Lemma 46).



(a)



(b)

$T[B]$

First part				Second part					Third part
v_1	v_2	v_3	v_4	e_1	e_2	e_3	e_4	e_5	ω
1	1	0	0	1	0	1	0	1	4
⋮				⋮					⋮

(c)

Figure 10: An example of $T[B]$: (a) a tree-decomposition (note that it is not nice), (b) a K_3 edge cover, (c) Table $T[B]$ and the row corresponding to the K_3 edge cover.

there is no edge between $B_\uparrow$ and $B_\downarrow$, $w \in B^+_\downarrow$ follows from Lemma 52, contradicting $X \notin \mathcal{X}_{B^+_\downarrow}$. □

Lemma 54. *If there is a solution $\mathcal{X}$, then there is a solution $\mathcal{X}'$ such that $|\mathcal{X}'| \leq |\mathcal{X}|$ and $|\{(v,w) \in E^-_{B^+_\downarrow,\mathcal{X}'} \mid w \in B_\downarrow\}| \in \{0,1\}$ for any bag B and any $v \in B$.*

Proof: For any solution $\mathcal{X}$, if there are $X = \{v,u,w\}, X' = \{v,u',w'\} \in \mathcal{X}$ with $v \in B, u, u' \in B_\uparrow$, $u \neq u'$, and $w, w' \in B_\downarrow$, then X and X' can be replaced with $Y = \{v,u,u'\}$ and $Y' = \{v,w,w'\}$, because there is no edge that is not covered by this replacement from Lemma 52. By applying this procedure as far as possible, the number of uncovered edges between v and $B_\downarrow$ becomes 0 or 1. □

Lemma 54 assures that each cell of the first part of each row is 1 or 0.

Lemma 55. *For two solutions $\mathcal{X}$ and $\mathcal{X}'$ and a bag B, assume that the signatures of the two partial solutions $\mathcal{X}_{B^+_\downarrow}$ and $\mathcal{X}'_{B^+_\downarrow}$ are the same. Then, there is a solution $\mathcal{X}''$ such that $\mathcal{X}''_{B^+_\downarrow} = \mathcal{X}_{B^+_\downarrow}$ and $|\mathcal{X}''| = |\mathcal{X}'| - |\mathcal{X}'_{B^+_\downarrow}| + |\mathcal{X}_{B^+_\downarrow}|$.*

Proof: If $E^-_{B^+_\downarrow,\mathcal{X}} = E^-_{B^+_\downarrow,\mathcal{X}'}$, the statement of this lemma holds by letting $\mathcal{X}'' = \mathcal{X}$ from Lemma 53. Therefore, we assume that $E^-_{B^+_\downarrow,\mathcal{X}} \neq E^-_{B^+_\downarrow,\mathcal{X}'}$ in the following part. Since $\mathcal{X}$ and $\mathcal{X}'$ have the same signature, every edge belonging to $E^-_{B^+_\downarrow,\mathcal{X}} - E^-_{B^+_\downarrow,\mathcal{X}'}$ or $E^-_{B^+_\downarrow,\mathcal{X}'} - E^-_{B^+_\downarrow,\mathcal{X}}$ has one end vertex in B and the other in $B_\downarrow$. Furthermore, for any $v \in B$, the number of edges belonging to $E^-_{B^+_\downarrow,\mathcal{X}} - E^-_{B^+_\downarrow,\mathcal{X}'}$ and incident to v is equal to the number of edges belonging to $E^-_{B^+_\downarrow,\mathcal{X}'} - E^-_{B^+_\downarrow,\mathcal{X}}$ and incident to v. Therefore, it is possible to give a one-to-one correspondence between these elements on the same edges (if exist). Assume that an element X' of $\mathcal{X}' - \mathcal{X}$ covers an edge (v,w) in $E^-_{B^+_\downarrow,\mathcal{X}'} - E^-_{B^+_\downarrow,\mathcal{X}}$ (where $v \in B$ and $w \in B_\downarrow$). In this case, X' is not a delta because if $X' = \{v,w,u\}$ is a delta, then from $X' \notin \mathcal{X}'_{B^+_\downarrow}$, $u \in B_\uparrow$ and $(u,v), (w,u) \in E$, but the existence of edge (w,u) contradicts Lemma 52. Moreover X' is not an I: otherwise, $X' = \{v,w\} \in \mathcal{X}'_{B^+_\downarrow}$, which contradicts $(v,w) \in E^-_{B^+_\downarrow,\mathcal{X}'}$. Therefore, X' is an L, and the edge covered in addition to (v,w) is (u,v) (where $u \in B_\uparrow$) without loss of generality. Let the edge in $E^-_{B^+_\downarrow,\mathcal{X}} - E^-_{B^+_\downarrow,\mathcal{X}'}$ corresponding to (v,w) be (v,w'). If we replace X' with $\{u,v,w'\}$, the signature does not change. Hence, if X of $\mathcal{X}$ covers an edge in $E^-_{B^+_\downarrow,\mathcal{X}'} - E^-_{B^+_\downarrow,\mathcal{X}}$, we can replace with X'. Let $\mathcal{X}'''$ be the set obtained by replacement. Here $|\mathcal{X}'''| = |\mathcal{X}' - \mathcal{X}|$ and X''' covers all edges in $E^-_{B^+_\downarrow,\mathcal{X}}$. Therefore, by letting $\mathcal{X}'' = \mathcal{X}''' \cup \mathcal{X}_B$, we obtain the desired solution. □

Lemma 56. *Let B be a forget node and B' be its child node. Let edge (u,w) satisfy*

$u \in B' - B$ and $w \in B'_{\downarrow}$. Let $\mathcal{X}$ be an optimal solution. Then, there is a set $X \in \mathcal{X}$ such that $u, w \in X$ and $X \subseteq B'^{+}_{\downarrow}$.

Proof: Assume that for every $X \in \mathcal{X}$ containing u and w, $X \nsubseteq B'^{+}_{\downarrow}$. Then there exists $v \in B'_{\uparrow}$ and $X \in \mathcal{X}$ such that $X = \{u, w, v\}$. Since B is a forget node, $B \subset B'$ and hence $v \in B_{\uparrow}$. On the other hand $(u, v) \in E$ or $(w, v) \in E$ must hold, because there is no I from Lemma 46, contradicting Lemma 52. □

Lemma 57. *Let B be a join node and B' and B'' be its child nodes. Then, $B'_{\downarrow} \cap B''_{\downarrow} = \emptyset$.*

Proof: From the definition of $B'_{\downarrow}$, any vertex belonging to $B'_{\downarrow}$ does not belong to B'. Therefore, from the definition of tree-decomposition (Condition (3) of Definition 47), these vertices do not belong to $B''_{\downarrow}$ either and hence this lemma follows. □

5.2.2 Algorithm

When creating table $T[B]$, if there are two or more partial solutions whose signatures are the same we can remove these partial solutions except one that has the minimum ω among them. A strict proof of the correctness of this operation will be shown after describing the algorithm (Lemma 58). We explain the algorithm step by step as follows. This algorithm scans the tree of a nice tree-decomposition in the postorder and creates a table $T[B]$ at each node (bag) B so that $T[B]$ has a partial optimal solution. Finally, an optimal solution exists in the table of the root node.

$T[B]$ is created from the tables of the children of B. Since the tree decomposition is nice, the number of children is at most two. The basic strategy of creating $T[B]$ is to enumerate all possible partial solutions. That is, for each row (partial solution) $\mathcal{X}$ in B', which is one of the children of B, the algorithm enumerates all possible cases of covering or uncovering the edges in $E^{-}_{B^{+}_{\downarrow},\mathcal{X}}$. if $|E^{-}_{B^{+}_{\downarrow},\mathcal{X}}|$ is bounded by a constant number, the enumeration for them can be done in constant time. However since $|E_{B^{+}_{\downarrow}}|$ grows up to m on the root of the decomposition tree, the number of different partial solutions becomes exponential. This problem is resolved by using Lemma 58. That is, from this lemma, keeping only partial solutions whose signatures are different is sufficient.

The operation for constructing $T[B]$ for each node (bag) B consists of two stages: one is Enumerating Stage and the other is Refinement Stage. In Enumerating Stage, all possible partial solutions are enumerated. In Refinement Stage, partial solutions that are unnecessary (i.e., that are not partial solutions of optimal solutions) are removed.

In the following, these stages are illustrated. First, Enumerating Stage for each type (leaf, introduce, forget, and join) is shown. Since Refinement Stage is the same for every type, it will be shown after explaining Enumerating Stages of all types.

Enumerating Stage

1. **Leaf nodes**

   Let B be a leaf node, and u be its (unique) element. Create Table as shown in **Table** 1.

Table 1: Since $G_B = (B = \{u\}, E_B = \emptyset)$ for leaf node $B = \{u\}$, $\mathcal{X}_B = \emptyset$ for any solution $\mathcal{X}$. Therefore, $T[B]$ looks like what is shown.

u	ω
0	0

2. **Introduce nodes**

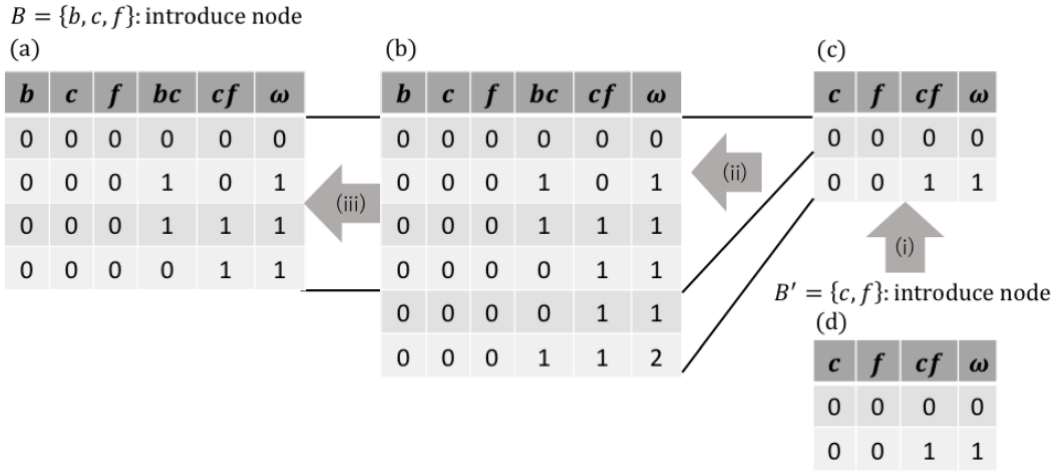
   Let $B = \{u, v_1, v_2, \ldots, v_k\}$ be an introduce node and $B' = \{v_1, v_2, \ldots, v_k\}$ be its child. For each row of $T[B']$, create rows of $T[B]$ according to the following operations. Let $\mathcal{X}_{B^+_\downarrow}$ be the partial solution corresponding to the row (of $T[B']$) (see Fig. 11).

   Step 1. For each edge (v_i, v_j), apply the following operations. Let w_i and w_j be vertices adjacent to v_i and v_j in $B'_\downarrow$, respectively, and are not covered by the partial solution that corresponds to the currently focused row of $T[B']$ yet (if exist). Note that v_i (resp., v_j) has at most one such vertex from Lemma 54. List up all possible combinations of the following cases and make a row corresponding to each of the combinations: (1) $\{v_i, v_j, u\}$ are covered by a K_3, (2) $\{v_i, v_j, w_i\}$ are covered by a K_3, and (3) $\{v_i, v_j, w_j\}$ are covered by a K_3. Create all combinations of the above three cases. Then, at most $2^3 = 8$ rows are created for the currently focused row in $T[B']$.

   Step 2. For each row created in Step 1, update the values in the cells of the first, the second and the third parts.

3. **Forget nodes**

   Let $B = \{v_1, v_2, \ldots, v_k\}$ be a forget node and $B' = \{u, v_1, v_2, \ldots, v_k\}$ be its child.



b	c	f	bc	cf	ω
0	0	0	0	0	0
0	0	0	1	0	1
0	0	0	1	1	1
0	0	0	0	1	1

(a)

b	c	f	bc	cf	ω
0	0	0	0	0	0
0	0	0	1	0	1
0	0	0	1	1	1
0	0	0	0	1	1
0	0	0	0	1	1
0	0	0	1	1	2

(b)

c	f	cf	ω
0	0	0	0
0	0	1	1

(c)

c	f	cf	ω
0	0	0	0
0	0	1	1

(d)

Figure 11: An example of making $T[B]$ from $T[B']$, where $B = \{b, c, f\}$ (introduce node) and $B' = \{c, f\}$ (the child of B), on the graph and the tree-decomposition shown in Fig. 9. Each row corresponds to a partial solution of the K_3 edge cover, e.g., the first row in Table (d) shows a partial solution $\{\emptyset\}$ and the second row shows a partial solution $\{\{c, f\}\}$. Furthermore, in Table (a), the first row shows a partial solution $\{\emptyset\}$ and the third row shows a partial solution $\{\{b, c, f\}\}$. Operation (i) refers to w_i and w_j in Step 1, and Step 1 corresponds to operations (i) and (ii). Operation (iii) is an operation included in the update of Step 2, which deletes rows that are no longer needed. (i) Since there is no non-zero cell in the first part of $T[B']$ in (d), the algorithm does nothing in (i). (ii) List up all possible cases to cover edges in B. In this case, we consider the combination of whether the two edges ((b, c) and (c, f)) are covered or not, i.e., the first row of (b) means neither (b, c) nor (c, f) are covered, the second row means (b, c) is covered but (c, f) is not, the third row means both (c, f) and (b, c) are covered by one K_3, and the fourth row means (c, f) is covered but (b, c) is not. (iii) Because the third row and the sixth row of (b) have the same signature and the value of ω of the latter is larger than the former, the latter (the sixth row) is deleted. Furthermore, because the fourth and the fifth rows are completely the same, only one of them is left. Consequently, the table of (a) is obtained.

For each row of $T[B']$, create $T[B]$ according to the following operations. Let $\mathcal{X}_{B^+_\downarrow}$ be the partial solution corresponding to the row (of $T[B']$) (see Fig. 12).

Step 1. For each edge (u, v_i), apply the following operations. Let w_u and w_i be vertices adjacent to u and v_i in $B'_\downarrow$, respectively, and are not covered by the partial solution that corresponds to the currently focused row of $T[B']$ yet (if exist). Note that u (resp., v_i) has at most one such vertex from Lemma 54. Let v_j be a vertex adjacent to at least one of u or v_i. List up all possible combinations of the following cases and make a row corresponding to each of the combinations: (1) $\{u, v_i, v_j\}$ are covered by a K_3, (2) $\{u, v_i, w_u\}$ are covered by a K_3, and (3) $\{u, v_i, w_i\}$ are covered by a K_3. Create all cases of combinations of the above three cases. Then, at most $2^3 = 8$ rows are created for the currently focused row in $T[B']$.

Step 2. For each row of $T[B]$, if edge (u, v_i) is not covered (i.e., the value of (u, v_i) is 0 in the second part of $T[B]$) and v_i has 1 in the cell of the first part of $T[B']$, $\{u, v_i, w_i\}$ are covered by a K_3 and 0 is set as the value of v_i in the first part of $T[B']$ (note that u does not exist in $B'_\uparrow$ from Lemma 56).

Step 3. For each row created in Step 1 and Step 2, update the values in the cells of the first, the second and the third parts.

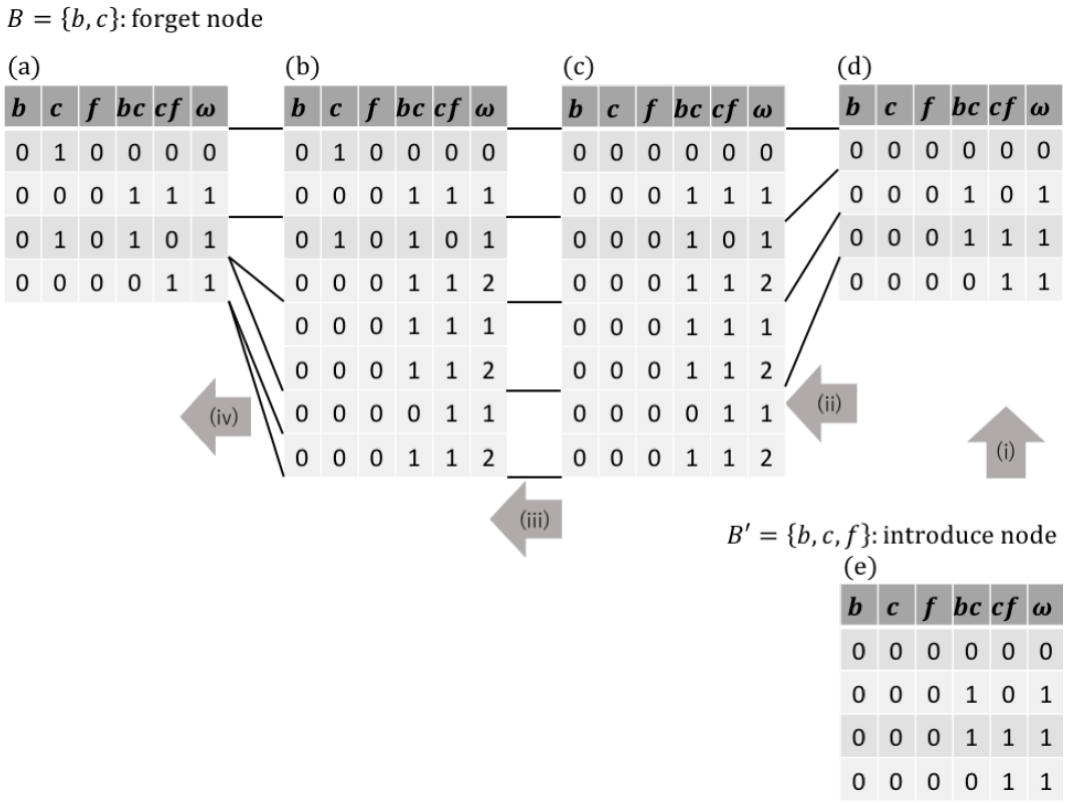
4. **Join nodes**

Let $B = \{v_1, v_2, \ldots, v_k\}$ be a join node and $B' = B'' = \{v_1, v_2, \ldots, v_k\}$ be its children. For each pair of a row in $T[B']$ and a row in $T[B'']$, create a row in $T[B]$ according to the following operations. Let $\mathcal{X}'_{B^+_\downarrow}$ and $\mathcal{X}''_{B^+_\downarrow}$ be the partial solutions corresponding to the row in $T[B']$ and the row in $T[B'']$, respectively. Create a row in $T[B]$, by applying the following operations to all pairs of rows in $T[B']$ and $T[B'']$.

Step 1. For each cell in the first part of the row, store the logical sum of the values of the corresponding cells of $\mathcal{X}'_{B^+_\downarrow}$ and $\mathcal{X}''_{B^+_\downarrow}$ (the correctness of this operation is supported by the fact $B'_\downarrow \cap B''_\downarrow = \emptyset$ proved in Lemma 57).

Step 2. For each cell in the second part of the row, put 1 if one of the values of the corresponding cells of $\mathcal{X}'_{B^+_\downarrow}$ and $\mathcal{X}''_{B^+_\downarrow}$ have 1, and 0 otherwise.

Step 3. For the cell in the third part of the row, store the sum of the values in



b	c	f	bc	cf	ω
0	1	0	0	0	0
0	0	0	1	1	1
0	1	0	1	0	1
0	0	0	0	1	1

b	c	f	bc	cf	ω
0	1	0	0	0	0
0	0	0	1	1	1
0	1	0	1	0	1
0	0	0	1	1	2
0	0	0	1	1	1
0	0	0	1	1	2
0	0	0	0	1	1
0	0	0	1	1	2

b	c	f	bc	cf	ω
0	0	0	0	0	0
0	0	0	1	1	1
0	0	0	1	0	1
0	0	0	1	1	2
0	0	0	1	1	1
0	0	0	1	1	2
0	0	0	0	1	1
0	0	0	1	1	2

b	c	f	bc	cf	ω
0	0	0	0	0	0
0	0	0	1	0	1
0	0	0	1	1	1
0	0	0	0	1	1

b	c	f	bc	cf	ω
0	0	0	0	0	0
0	0	0	1	0	1
0	0	0	1	1	1
0	0	0	0	1	1

Figure 12: An example of making $T[B]$ from $T[B']$, where $B = \{b, c\}$ (forget node) and $B' = \{b, c, f\}$ (the child of B), on the graph and the tree-decomposition shown in Fig. 9. Each row corresponds to a partial solution of the K_3 edge cover, e.g., the first row in table (d) shows a partial solution $\emptyset$ and the second row shows a partial solution $\{\{b, c, f\}\}$. Operation (i) refers to w_i and w_j in Step 1, and Step 1 corresponds to operations (i) and (ii). Step 2 does not apply to this node. Operation (iii) and (iv) is an operation included in the update of Step 3, Operation (iii) is the edge that disappeared in the forget node, Operation (iv) deletes rows that are no longer needed. (i) Since there is no non-zero cell in the first part of $T[B']$ in (e), the algorithm does nothing in (i). (ii) List up all possible cases to cover edges in B. In this case, we consider the combination of whether the two edges ((b, c) and (c, f)) are covered or not, i.e., the first row of (c) means neither (b, c) nor (c, f) are covered, the second row means (b, c) and (c, f) are covered, the third row means (c, f) is covered but (b, c) is not. (iii) The value of the column corresponding to vertex c (the second column) is set to 1, because edge (c, f) and f do not exist in B. (iv) Because the second, the fourth, the fifth, the sixth and the eighth rows of (b) have the same signature and the value of ω of the second is the smallest, the fourth, the fifth, the sixth and the eighth rows are deleted. Consequently, the table of (a) is obtained.

the cells of the third part of $\mathcal{X}'_{B^+_\downarrow}$ and $\mathcal{X}''_{B^+_\downarrow}$.

After Steps 1 to 3, apply the following operations.

Step 4. For each v_i, if both of the corresponding cells of $T[B']$ and $T[B'']$ are 1 (and the corresponding cell in $T[B]$ is 0 as a result of the logical summation), then increase the value of the row in the third part of $T[B]$ by 1.

Refinement Stage

After finishing creating columns of $T[B]$ by the above operations, if there are two or more partial solutions (rows) that have the same signature in $T[B]$, then leave only one partial solution that has the minimum ω among them (i.e., delete the others). The algorithm finally outputs the minimum size of K_3 edge cover in wide sense and its covering.

We prove the correctness and the computation time as follows.

Lemma 58. *If $T[B]$ includes a partial solution of an optimal solution, then at least one of the kept solutions remains after Refinement Stage.*

Proof: Assume that a partial solution of an optimal solution $\mathcal{X}^*$ was deleted in Refinement Stage of node B. We denote the deleted optimal partial solution by $\mathcal{X}^*_{B^+_\downarrow}$. A partial solution $\mathcal{X}_{B^+_\downarrow}$ that has the same signature must remain. From the rule of Refinement Stage,

$$|\mathcal{X}_{B^+_\downarrow}| \leq |\mathcal{X}^*_{B^+_\downarrow}|. \tag{26}$$

By regarding $\mathcal{X}^*$ and $\mathcal{X}$ as $\mathcal{X}'$ and $\mathcal{X}$, respectively, of Lemma 55, $\mathcal{X}''$ constructed in the proof of Lemma 55 is also a solution. From (26), $|\mathcal{X}''| = |\mathcal{X}^*| - |\mathcal{X}^*_{B^+_\downarrow}| + |\mathcal{X}_{B^+_\downarrow}| \leq |\mathcal{X}^*|$. Thus $\mathcal{X}''$ is also an optimal solution. $\mathcal{X}_{B^+_\downarrow}$ can be regarded as a partial solution of $\mathcal{X}''$. □

Lemma 59. *For any bag B, the partial solution stored in $T[B]$ contains the signature of an optimal partial solution.*

Proof: This can be easily proved by induction. In the leaf node, it is clearly trivial. We assume that at the child nodes, the optimal solution is contained. The algorithm makes all possible combinations at the parent node (introduce, forget, join). Hence, the optimal solution is contained in it. □

Now we establish the proof of Theorem 4 as follows.

Proof of Theorem 4: From Lemma 59, since an optimal solution exists in the solution

stored in the root node, the algorithm correctly gives the optimal solution. Thus we estimate the computation time. First we calculate the number of columns in the table $T[B]$ of a bag B. The number of columns in the first part, which is equal to the number of the vertices in the bag, is at most $t + 1$, the number of columns in the second part, which is equal to the number of edges in the bag, is at most $\binom{t+1}{2} = t(t+1)/2$ and there is another column for the third part. Thus the number of columns of a table is at most $t + 1 + t(t + 1)/2 + 1 = O(t^2)$. Next, we calculate the number of rows. The maximum number of rows in a table after finishing Refinement Stage is equal to the maximum possible variations of signatures. Since there are at most 2^{t+1} variations for the first part and $2^{t(t+1)/2}$ variations for the second part, the total number of possible variations is at most $2^{t+1}2^{t(t+1)/2} = 2^{(t+1)(t+2)/2}$, which is an upper bound of the number of rows of a table after finishing the Refinement Stage. However, in Enumerating Stage, more rows can be created. In an introduce or forget node, we may create at most 8 rows for each row of the table of the child node. Thus a total of $8 \cdot 2^{(t+1)(t+2)/2}$ rows may be created in Enumerating Stage of an introduce or forget node. In a join node, at most $(2^{(t+1)(t+2)/2})^2 = 2^{(t+1)(t+2)}$ rows are created in Enumerating Stage. By comparing $8 \cdot 2^{(t+1)(t+2)/2}$ and $2^{(t+1)(t+2)}$, it follows that the number of rows are $O(2^{(t+1)(t+2)})$.

In Refinement Stage we compare all pairs of rows. Hence we compare $O((2^{(t+1)(t+2)})^2) = O(2^{2(t+1)(t+2)})$ rows in a node. Comparing one pair of rows requires $O(t^2)$ time, since the number of columns is $O(t^2)$. Therefore the computation time required in a node is $O(2^{2(t+1)(t+2)}t^2)$.

Finally, since the number of nodes of the tree is $O(n)$, it follows that the total computation time is $O(2^{2(t+1)(t+2)}t^2n)$. □

Chapter 6 Conclusion

We worked on two techniques. First, we created a general framework for sublinear progressive algorithms. By proving the SPA Theorem, we showed that a sublinear progressive algorithm can be constructed from any problem that has both a constant-time algorithm and an exact algorithm. This made it possible to convert any constant-time algorithm into a sublinear progressive algorithm without losing the computation time in the big-O sense. This is the first method to convert a sublinear-time algorithm into a progressive algorithm, and the first research that shows a general construction method for progressive algorithms. Furthermore, we clarified the general relationship between approximation parameter, fault probability, and computation time in constant-time algorithms. We have developed a theoretical basis for a new algorithm with a high affinity for large-scale data.

Second, we constructed an FPT algorithm for a combinatorial optimization problem that has important applications in the design of experiments. We considered the problem including the "spilling-out" corresponding to the combination of "don't care" in comparative experiments. We call this problem the K_k edge cover problem in a wide sense. We found that the K_3 edge cover problem in a wide sense is NP-complete even for planar and cubic graphs. We also showed FPT algorithms for several parameters: one is an $O(|E||V| + 2^k|E|)$-time algorithm and the other is an $O(2^{2(t+1)(t+2)}t^2n)$-time algorithm, where k is the number of 3-cliques and t is the tree-width (under the assumption that a tree-decomposition of width t is given). If $k \geq 4$, a k-clique can cover the non-adjacent edges. A set of four vertices $\{a, b, c, d\}$ can be covered by a 4-clique even if the induced subgraph by them is not a clique. Therefore, the result for $k = 3$ cannot be simply extended to $k \geq 4$.

In recent years, the rapid increase in data size has greatly exceeded the rate of evolution of hardware. We are sometimes faced with the problem that large-scale problems are difficult to solve by using the traditional algorithms. By addressing our two problems, we may make waves in creating fast algorithms for the sublinear computation paradigm. Linear and sublinear-time algorithms that break the traditional notion of "efficient" are likely to become increasingly important in the near future. Our target is to expand this research and develop new algorithms for big data that will

support innovation in the big data era.

References

[1] Faisal Abu-khzam, Rebecca Collins, Michael Fellows, Michael Langston, W. Suters, and Christopher Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In Proc. of *the 6th Workshop on Algorithm Engineering and Experiments (ALENEX) and the First Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pp. 62–69, New Orleans, LA, USA, Jan 2004.

[2] Faisal Abu-khzam, Michael Langston, Pushkar Shanbhag, and Christopher Symons. Scalable Parallel Algorithms for FPT Problems. *Algorithmica*, Vol. 45, pp. 269–284, Jul 2006.

[3] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, USA, May 2005.

[4] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In Proc. of *the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pp. 309–322, NY, USA, Jan 2008.

[5] Jochen Alber, Hans L. Bodlaender, Henning Fernau, Ton Kloks, and Rolf Niedermeier. Fixed parameter algorithms for dominating set and related problems on planar graphs. *Algorithmica*, Vol. 33, pp. 461–493, Aug 2002.

[6] Jochen Alber, Frederic Dorn, and Rolf Niedermeier. Experimental evaluation of a tree decomposition-based algorithm for vertex cover on planar graphs. *Discrete Applied Mathematics*, Vol. 145, No. 2, pp. 219–231, Jan 2005.

[7] Jochen Alber, Henning Fernau, and Rolf Niedermeier. Parameterized complexity: exponential speed-up for planar graph problems. *Journal of Algorithms*, Vol. 52, No. 1, pp. 26–56, Jul 2004.

[8] Sander P. A. Alewijnse, Timur M. Bagautdinov, Mark de Berg, Quirijn W. Bouts, Alex P. Ten Brink, Kevin Buchin, and Michel A. Westenberg. Progressive geometric algorithms. In Proc. of *the 30th Annual Symposium on Computational Geometry (SoCG 2014)*, Vol. 6, pp. 72–92, Jun 2014.

[9] Noga Alon, Richard A. Duke, Ralph H. Lefmann, Vojtěch Rödl, and Raphael Yuster. The algorithmic aspects of the regularity lemma. *Journal of Algo-*

rithms, Vol. 16, No. 1, pp. 80–109, Jan 1994.

[10] Noga Alon, Eldar Fischer, Michael Krivelevich, and Mario Szegedy. Efficient testing of large graphs. *Combinatorica*, Vol. 20, pp. 451–476, Apr 2000. In Proc. of *the 40th Annual Symposium on Foundations of Computer Science (FOCS 1999)*, New York, NY, USA, pp. 656–666, Oct 1999.

[11] Noga Alon, Eldar Fischer, Ilan Newman, and Asaf Shapira. A combinatorial characterization of the testable graph properties: It's all about regularity. *SIAM Journal on Computing*, Vol. 39, pp. 143–167, Jan 2009. In Proc. of *the 38th Annual ACM Symposium on Theory of Computing (STOC 2006)*, Seattle, WA, USA, pp.251–260, May 2006.

[12] Noga Alon and Asaf Shapira. A characterization of the (natural) graph properties testable with one-sided error. *SIAM Journal on Computing*, Vol. 37, No. 6, pp. 1703–1727, Mar 2008. In Proc. of the 46th Annual IEEE Symposium on Foundations of Computer Science *(FOCS 2005)*, Pittsburgh, PA, USA, pp. 429–438, Oct 2005.

[13] Noga Alon and Asaf Shapira. Every monotone graph property is testable. *SIAM Journal on Computing*, Vol. 38, No. 2, pp. 505–522, Apr 2008. In Proc. of *the 37th Annual ACM Symposium on Theory of Computing (STOC 2005)*, Baltimore, MD, USA, pp. 128–138, May 2005.

[14] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, Vol. 8, No. 2, pp. 277–284, Apr 1987.

[15] Jasine Babu, Areej Khoury, and Ilan Newman. Every property of outerplanar graphs is testable. In Proc. of *the 19th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX 2016), and the 20th International Workshop on Randomization and Computation (RANDOM 2016)*, Vol. 60, pp. 21:1–21:19, Sep 2016.

[16] Zongwen Bai, Jianhua Tu, and Yongtang Shi. An improved algorithm for the vertex cover P_3 problem on graphs of bounded treewidth. *Discrete Mathematics & Theoretical Computer Science*, Vol. 21, No. 4, Nov 2019.

[17] R. Balasubramanian, Michael R. Fellows, and Venkatesh Raman. An improved fixed-parameter algorithm for vertex cover. *Information Processing Letters*,

Vol. 65, No. 3, pp. 163–168, Feb 1998.

[18] Nicolas Barnier and Pascal Brisset. Solving the Kirkman's schoolgirl problem in a few seconds. *Constraints*, Vol. 10, No. 1, pp. 7–21, Jan 2005. In Proc. of *the 8th International Conference on Principles and Practice of Constraint Programming (2002)*, Ithaca, NY, USA, pp. 477–491, Sep 2002.

[19] Tuğkan Batu, Petra Berenbrink, and Christian Sohler. A sublinear-time approximation scheme for bin packing. *Theoretical Computer Science*, Vol. 410, No. 47–49, pp. 5082–5092, Nov 2009.

[20] Itai Benjamini, Oded Schramm, and Asaf Shapira. Every minor-closed property of sparse graphs is testable. *Advances in Mathematics*, Vol. 223, No. 6, pp. 2200–2218, Apr 2010. In Proc. of *the 40th Annual ACM Symposium on Theory of Computing (STOC 2008)*, Victoria, British Columbia, Canada, pp. 393–402, May 2008.

[21] Arnab Bhattacharyya and Yuichi Yoshida. *Property Testing — Problems and Techniques*. Springer, Feb 2022.

[22] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, Vol. 47, No. 3, pp. 549–595, Dec 1993.

[23] Mark Boddy and Thomas L. Dean. Solving time-dependent planning problems. In Proc. of *the 11th international joint conference on Artificial intelligence (IJCAI 1989)*, Vol. 2, pp. 979–984. Brown University, Department of Computer Science, Aug 1989.

[24] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In Proc. of *the 25th Annual ACM Symposium on Theory of Computing (STOC 1993)*, STOC '93, pp. 226–234, New York, NY, USA, 1993. Association for Computing Machinery.

[25] Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michał Pilipczuk. An $O(c^k n)$ 5-approximation algorithm for treewidth. *SIAM Journal on Computing*, Vol. 45, No. 2, pp. 317–378, 2016. In Proc. of *the 54th Annual Symposium on Foundations of Computer Science (FOCS 2013)*, Washington, DC, USA, pp. 499–508, Oct 2013.

[26] Andrej Bogdanov, Kenji Obata, and Luca Trevisan. A lower bound for testing

3-colorability in bounded-degree graphs. In Proc. of *the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2002)*, pp. 93–102, Nov 2002.

[27] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, New York, NY, USA, 1998.

[28] Liming Cai and David Juedes. On the existence of subexponential parameterized algorithms. *Journal of Computer and System Sciences*, Vol. 67, No. 4, pp. 789–807, Dec 2003.

[29] P. J. Cameron and J. H. van Lint. *Graph Theory, Coding Theory and Block Designs*, pp. 1–10. London Mathematical Society Lecture Note Series. Cambridge University Press, Sep 1975.

[30] Magnus Carlsson. Monads for incremental computing. In Proc. of *the 7th ACM SIGPLAN international conference on Functional programming (ICFP 2002)*, Vol. 37, pp. 26–35, Sep 2002.

[31] L. Sunil Chandran and Fabrizio Grandoni. Refined memorization for vertex cover. *Information Processing Letters*, Vol. 93, No. 3, pp. 123–131, Feb 2005. In Proc. of *1st International Workshop on Parameterized and Exact Computation (IWPEC 2004)*, Bergen, Norway, pp. 61–70, Sep 2004.

[32] Gary Chartrand and Ping Zhang. *A First Course in Graph Theory*. Courier Corporation, Washington, DC, USA, May 2013.

[33] James Cheetham, Frank Dehne, Andrew Rau-Chaplin, Ulrike Stege, and Peter Taillon. Solving large FPT problems on coarse-grained parallel machines. *Journal of Computer and System Sciences*, Vol. 67, pp. 691–706, Dec 2003.

[34] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, Vol. 41, No. 2, pp. 280–301, Nov 2001.

[35] Jianer Chen, Iyad Kanj, and Ge Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, Vol. 411, No. 40–42, pp. 3736–3756, Sep 2010.

[36] Herman Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, Vol. 23, No. 4, pp. 493–507, Dec 1952.

[37] Kyohei Chiba, Rémy Belmonte, Hiro Ito, Michael Lampis, Atsuki Nagao, and Yota Otachi. k_3 edge cover problem in a wide sense. *Journal of Information Processing*, Vol. 28, pp. 849–858, Dec. 2020.

[38] Kyohei Chiba, Mikiya Imura, Hiro Ito, and Ilan Newman. Sublinear progressive algorithms – the framework and fundamental theorems, *The 5th International Workshop on Innovative Algorithms for Big Data (IABD2019)*, Kyoto, Japan, Nov 2019.

[39] Kyohei CHIBA and Hiro ITO. Sublinear computation paradigm: Constant-time algorithms and sublinear progressive algorithms. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E105-A, No. 3, Mar. 2022.

[40] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, Vol. 14, No. 1, pp. 210–223, 1985.

[41] Giraud-Carrier Christophe. A Note on the Utility of Incremental Learning. *AI Communications*, Vol. 13, No. 4, pp. 215–233, Feb 2000.

[42] Charles J. Colbourn and Jeffrey H. Dinitz. *Handbook of Combinatorial Designs, Second Edition (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, Boca Raton, FL, USA, Nov 2006.

[43] Marek Cygan, Marcin Pilipczuk, and Michał Pilipczuk. Known Algorithms for Edge Clique Cover are Probably Optimal. *SIAM Journal on Computing*, Vol. 45, No. 1, pp. 67–83, Jan 2016.

[44] Ankan Kumar Das. On a few progressive algorithms. Master's thesis, Indian Statistical Institute, 203 BT Road, Kolkata, Jul 2020.

[45] Jane W. Di Paola. Block designs and graph theory. *Journal of Combinatorial Theory*, Vol. 1, No. 1, pp. 132–148, Jun 1966.

[46] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, Aug 2005.

[47] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[48] Henning Fernau. On parameterized enumeration. In Proc. of *the 8th Annual International Conference on Computing and Combinatorics (COCOON 2002)*, pp. 151–179, Singapore, Aug 2002.

[49] Hendrik Fichtenberger, Pan Peng, and Christian Sohler. Every testable (infinite) property of bounded-degree graphs contains an infinite hyperfinite subproperty. In Proc. of *the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2019)*, pp. 714–726, Jan 2019.

[50] Denis Firsov and Wolfgang Jeltsch. Purely functional incremental computing. In Proc. of *The Brazilian Symposium on Programming Languages (SBLP)*, Vol. 9889, pp. 62–77, Sep 2016.

[51] Fedor V. Fomin, Ioan Todinca, and Yngve Villanger. Large Induced Subgraphs via Triangulations and CMSO. *SIAM Journal on Computing*, Vol. 44, No. 1, pp. 54–87, Feb 2015. In Proc. of *the 25 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2014)*, Portland, OR, USA, pp. 582–583, Jan 2014.

[52] Michael R. Garey and David S. Johnson. The rectilinear Steiner tree problem is NP-complete. *SIAM Journal on Applied Mathematics*, Vol. 32, No. 4, pp. 826–834, Jun 1977.

[53] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, New York, NY, USA, 1979.

[54] Fănică Gavril. Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph. *SIAM Journal on Computing*, Vol. 1, No. 2, pp. 180–187, Jun 1972.

[55] Fănică Gavril. Algorithms on circular-arc graphs. *Networks*, Vol. 4, No. 4, pp. 357–369, Jan 1974.

[56] Lior Gishboliner and Asaf Shapira. Deterministic vs Non-deterministic Graph Property Testing. *Israel Journal of Mathematics*, Vol. 204, pp. 397–416, Apr 2013.

[57] Oded Goldreich. *Property Testing - Current Research and Surveys*, Vol. 6390 of *Lecture Notes in Computer Science*. Springer, Jan 2010.

[58] Oded Goldreich. *Introduction to Property Testing*. Cambridge University Press, New York, NY, USA, Nov 2017.

[59] Oded Goldreich, Shari Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *Journal of the ACM*, Vol. 45, No. 4, pp. 653–750, Jul 1998.

[60] Oded Goldreich and Dana Ron. Property testing in bounded degree graphs. *Algorithmica*, Vol. 32, No. 2, pp. 302–343, Feb 2002. In Proc. of *the 29th Annual ACM Symposium on Theory of Computing (STOC 1997)*, El Paso, TX, USA, pp. 406–415, May 1997.

[61] Martin Charles Golumbic. The complexity of comparability graph recognition and coloring. *Computing*, Vol. 18, No. 3, pp. 199–208, Sep 1977.

[62] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Data reduction and exact algorithms for clique cover. *ACM Journal of Experimental Algorithmics (JEA)*, Vol. 13, pp. 2.2–2.15, Feb 2009.

[63] Duc A. Hoang, Akira Suzuki, and Tsuyoshi Yagita. Reconfiguring k-path vertex covers. In Proc. of *the 14th International Conference and Workshop on Algorithms and Computation (WALCOM 2020)*, pp. 133–145. Springer, Feb 2020.

[64] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, Vol. 58, No. 301, pp. 13–30, Mar 1963.

[65] Hiro Ito. Constant-time algorithms for complex networks. In *2016 3rd Asia-Pacific World Congress on Computer Science and Engineering (APWC on CSE)*, pp. 10–17, 2016.

[66] Hiro Ito. Every property is testable on a natural class of scale-free multigraphs. In Proc. of *the 24th Annual European Symposium on Algorithms (ESA 2016)*, Vol. 57, pp. 51:1–51:12, 2016.

[67] Hiro Ito. What graph properties are constant-time testable? — dense graphs, sparse graphs, and complex networks. *The Review of Socionetwork Strategies*, Vol. 13, pp. 101–121, Sep 2019.

[68] Hiro Ito and Kazuo Iwama. Enumeration of isolated cliques and pseudo-cliques. *ACM Transactions on Algorithms (TALG)*, Vol. 5, No. 4, pp. 1–21, Oct 2009.

[69] Hiro Ito, Kazuo Iwama, and Tsuyoshi Osumi. Linear-time enumeration of isolated cliques. In Proc. of *the 13th Annual European Symposium on Algorithms (ESA 2005)*, pp. 119–130. Springer, 2005.

[70] Hiro Ito, Areej Khoury, and Ilan Newman. On the characterization of 1-sided

error strongly testable graph properties for bounded-degree graphs. *Computational Complexity*, Vol. 29, pp. 1–45, Jan 2020.

[71] Hiro Ito, Susumu Kiyoshima, and Yuichi Yoshida. Constant-time approximation algorithms for the knapsack problem. In Proc. of *the 9th Annual international conference on Theory and Applications of Models of Computation (TAMC 2012)*, pp. 131–142, May 2012.

[72] Hiro Ito, Atsuki Nagao, and Teagun Park. Generalized shogi, chess, and xiangqi are constant-time testable. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E102.A, No. 9, pp. 1126–1133, Sep 2019.

[73] Richard M. Karp. Reducibility among combinatorial problems. In Proc. of *Complexity of Computer Computations*, pp. 85–103. Springer, Mar 1972.

[74] Richard M. Karp. On-line algorithms versus off-line algorithms: How much is it worth to know the future? In Proc. of *the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92*, Vol. 1 - Vol. I, pp. 416–429, Sep 1992.

[75] Petteri Kaski and Patric RJ Östergård. There are exactly five biplanes with $k = 11$. *Journal of Combinatorial Designs*, Vol. 16, No. 2, pp. 117–127, Jun 2008.

[76] Naoki Katoh, Yuya Higashikawa, Hiro Ito, Shun Kataoka, Takuya Kida, Toshiki Saitoh, Tetsuo Shibuya, Kazuyuki Tanaka, and Yushi Uno. Preface for the special issue on the project "foundation of innovative algorithms for big data". *The Review of Socionetwork Strategies*, Vol. 13(2), pp. 99–100, Oct 2019.

[77] Naoki Katoh, Yuya Higashikawa, Hiro Ito, Atsuki Nagao, Tetsuo Shibuya, Adnan Sljoka, Kazuyuki Tanaka, and Yushi Uno. *Sublinear Computation Paradigm: Algorithmic Revolution in the Big Data Era*. Springer, Singapore, Nov 2021.

[78] Thomas P. Kirkman. *The Lady's and Gentleman's Diary*, p. 48. Company of Stationers, 1850.

[79] Ton Kloks. *Treewidth: computations and approximations*, Vol. 842. Springer Science & Business Media, Aug 1994.

[80] Mitsuru Kusumoto and Yuichi Yoshida. Testing forest-isomorphism in the adjacency list model. In Proc. of *the 41st International Colloquium on Automata, Languages, and Programming (ICALP 2014)*, Vol. 8572, pp. 763–774. Springer, 2014.

[81] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. Efficient and progressive group steiner tree search. In Proc. of *the 2016 International Conference on Management of Data (SIGMOD 2016)*, pp. 91–106. Association for Computing Machinery, Jun 2016.

[82] László Lovász and Katalin Vesztergombi. Non-deterministic graph property testing. *Combinatorics, Probability and Computing*, Vol. 22, No. 5, pp. 749–762, Jul 2013.

[83] Jiří Matoušek and Jaroslav Nešetřil. *Invitation to Discrete Mathematics*. Oxford University Press, New York, NY, USA, Sep 1998.

[84] Amir Mesrikhani and Mohammad Farshi. Progressive sorting in the external memory model. *The CSI Journal on Computer Science and Engineering*, Vol. 15, No. 2, 2018. In Proc. of *the 48th Annual Iranian Mathematics Conference (AIMC48)*, Hamedan, Iran, Aug 2017.

[85] Amir Mesrikhani, Mohammad Farshi, and Mansoor Davoodi. Progressive algorithm for euclidean minimum spanning tree. In Proc. of *the 1st Iranian Conference on Computational Geometry (ICCG 2018)*, pp. 29–32, 2018.

[86] Amir Mesrikhani, Mohammad Farshi, and Behnam Iranfar. A progressive algorithm for the closest pair problem. *International Journal of Computer Mathematics: Computer Systems Theory*, Vol. 6, No. 2, pp. 130–136, Apr 2021.

[87] Michael Mitzenmacher and Eli Upfal. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, New York, NY, USA, Jan 2005.

[88] Eiji Miyano, Toshiki Saitoh, Ryuhei Uehara, Tsuyoshi Yyagita, and Tom C. van der Zanden. Complexity of the maximum k-path vertex cover problem. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E103.A, No. 10, pp. 1193–1201, Oct 2020.

[89] Ilan Newman and Christian Sohler. Every property of hyperfinite graphs is

testable. *SIAM Journal on Computing*, Vol. 42, No. 3, pp. 1095–1112, Jun 2013.

[90] Rolf Niedermeier and Peter Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, Vol. 73, No. 3–4, pp. 125–129, Feb 2000.

[91] Rolf Niedermeier and Peter Rossmanith. On efficient fixed-parameter algorithms for weighted vertex cover. *Journal of Algorithms*, Vol. 47, No. 2, pp. 63–77, Jan 2003.

[92] Mohammad Noshad and Maite Brandt-Pearce. Expurgated ppm using symmetric balanced incomplete block designs. *IEEE Communications Letters*, Vol. 16, No. 7, pp. 968–971, Jul 2012.

[93] James Orlin. Contentment in graph theory: Covering graphs with cliques. In Proc. of *Indagationes Mathematicae*, Vol. 80, pp. 406–424. Elsevier, 1977.

[94] ABD14 Project. Foundations of innovative algorithms for big data. `http://crest-sublinear.jp/en/`.

[95] Johannes Ringmark. A case study for progressive algorithms - an investigation into progressive extraction of intermediate solutions for the weighted interval scheduling problem. Master's thesis, University of Gothenburg, Gothenburg, Sweden, Oct 2020.

[96] Ronitt Rubinfeld and Madhu Sudan. Robust characterizations of polynomials withapplications to program testing. *SIAM Journal on Computing (SICOMP)*, Vol. 25, No. 2, pp. 252–271, Feb 1996.

[97] Herbert J. Ryser. *Combinatorial Mathematics*, pp. 96–102. American Mathematical Society, 1963.

[98] Amir Saffari, Christian Leistner, Jakob Santner, Martin Godec, and Horst Bischof. On-line random forests. In Proc. of *the 12th International Conference on Computer Vision Workshops (ICCV Workshops 2009)*, pp. 1393–1400, Nov 2009.

[99] Alexa Megan Sharp. *Incremental Algorithms: Solving Problems in a Changing World*. PhD thesis, Cornell University, Ithaca, NY 14850, USA, Aug 2007.

[100] Takeya Shigezumi, Yushi Uno, and Osamu Watanabe. A new model for a scale-free hierarchical structure of isolated cliques. *Journal of Graph Algorithms and*

Applications (JGAA), Vol. 15, No. 5, pp. 661–682, Feb 2011. In Proc. of *the 4th International Conference and Workshop on Algorithms and Computation (WALCOM 2010)*, Dhaka, Bangladesh, pp. 216–227, Feb 2010.

[101] Steven Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Addison-Wesley, Boston, MA, USA, Jan 1990.

[102] Douglas R. Stinson. *Combinatorial Designs: Constructions and Analysis*. Springer, 2004.

[103] Anne P. Street and Deborah J. Street. *Combinatorics of Experimental Design*. Oxford Science Publications, 1987.

[104] Endre Szemerédi. Regular partitions of graphs. Technical report, Stanford University, Stanford, CA 94305, USA, 1975.

[105] Jianhua Tu, Lidong Wu, Jing Yuan, and Lei Cui. On the vertex cover p_3 problem parameterized by treewidth. *Journal of Combinatorial Optimization*, Vol. 34, No. 2, pp. 414–425, Aug 2017.

[106] Richard M. Wilson. An existence theory for pairwise balanced designs I: Composition theorems and morphisms. *Journal of Combinatorial Theory, Series A*, Vol. 13, No. 2, pp. 220–245, 1972.

[107] Andrew Chi-Chih Yao. Lower bounds to randomized algorithms for graph properties. *Journal of Computer and System Sciences*, Vol. 42, No. 3, pp. 267–287, Jun 1991. In Proc. of *the 28th Annual Symposium on Foundations of Computer Science (SFCS 1987)*, Los Angeles, CA, USA, pp. 393-400, Oct 1987.

[108] Yuichi Yoshida. A characterization of locally testable affine-invariant properties via decomposition theorems. In Proc. of *the 46th Annual ACM Symposium on Theory of Computing (STOC 2014)*, pp. 154–163, New York, NY, USA, 2014.

[109] Yuichi Yoshida and Hiro Ito. Query-number preserving reductions and linear lower bounds for testing. *IEICE Transactions on Information and Systems*, Vol. E93.D, No. 2, pp. 233–240, Feb 2010.

[110] Yuichi Yoshida and Hiro Ito. Property testing on k-vertex-connectivity of graphs. *Algorithmica*, Vol. 62, pp. 701–712, Apr 2012. In Proc. of *the 35th*

International Colloquium on Automata, Language and Programming (ICALP 2008), Reykjavik, Iceland, pp. 539–550, Jul 2008.

[111] Zhi-Hua Zhou and Zhao-Qian Chen. Hybrid decision tree. *Knowledge-Based Systems*, Vol. 15, No. 8, pp. 515–528, Nov 2002.

[112] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, Vol. 17, No. 3, p. 73, Mar 1996.

List of Publications

Refereed Journals

Kyohei Chiba, Rémy Belmonte, Hiro Ito, Michael Lampis, Atsuki Nagao, Yota Otachi. K_3 Edge Cover Problem in a Wide Sense, Journal of Information Processing, 2020, Vol. 28, pp. 849–858, Dec. 2020.

Kyohei Chiba and Hiro Ito. Sublinear Computation Paradigm: Constant-Time Algorithms and Sublinear Progressive Algorithms, IEICE Transactions, Vol. E105-A, No. 3, Mar. 2022.

International Conferences

Kyohei Chiba, Hiro Ito and Atsuki Nagao. K_3 edge cover in a wide sense, The 20th Japan Conference on Discrete and Computational Geometry, Graphs, and Games (JCDCG2017), Tokyo University of Science, Tokyo, Japan, Sep. 2017.

Kyohei Chiba, Mikiya Imura, Hiro Ito and Ilan Newman. Sublinear progressive algorithms – The framework and fundamental theorems, The 5th International Workshop on Innovative Algorithms for Big Data (IABD2019), Kyoto University, Kyoto, Japan, Nov. 2019.

Kyohei Chiba, Mikiya Imura, Hiro Ito and Ilan Newman. The framework of sublinear progressive algorithms, Singapore Global Young Scientists Summit, Matrix, Biopolis, Singapore, Jan. 2020.

Domestic Conferences

Kyohei Chiba, Rémy Belmonte, Hiro Ito, Atsuki Nagao. K_3 edge cover in a wide sense, COMP, The Institute of Electronics, Information and Communication Engineers, Aomori Prefecture Tourist Center, Aug.2017.

Kyohei Chiba, Hiro Ito, Atsuki Nagao. K_3 edge cover in a wide sense, IEICE General Conference 2020, Tokyo Denki University, Tokyo, Japan, Mar. 2018.

Kyohei Chiba, Mikiya Imura, Hiro Ito and Ilan Newman. The framework of sublinear progressive algorithms, WINTER FESTA Episode 5, Hitotsubashi Hall, Tokyo, Japan, Dec. 2020.