國 立 中 正 大 學

資訊工程研究所博士論文

固定參數演算法與性質測試之研究

A Study on Fixed-Parameter Algorithms and Property Testing

研究生：　林莊傑　撰

指導教授：張貿翔　博士

Dr. Peter Rossmanith

中 華 民 國 一 百 年 七 月

Agreement of Joint Advisement of a Ph.D. Dissertation

Department of Computer Science and Information Engineering
National Chung Cheng University

Student's Name: Chuang-Chieh Lin

Principal Advisor's Name: Dr. Maw-Shang Chang

Co-Advisor's Name: Dr. Peter Rossmanith

This is an agreement of jointly advising the Ph.D. candidate, Chuang-Chieh Lin, for his dissertation.

Student's Signature: Chuang-Chieh Lin 林莊傑 Date: 20 Oct. 2010

Principal Advisor's Signature: M.S. Chang Date: 21 Oct. 2010

Co-Advisor's Signature: P. Rossmanith Date: 6 Nov 2010

Chair's Signature: Jiun-Ln Guan Date: 8 Dec. 2010.

國立中正大學博士班研究生
學位考試同意書

本人所指導 資訊工程學系

研究生 林莊傑 所提之論文

A Study on Fixed-Parameter Algorithms and Property Testing

同意其提付 博 士學位論文考試

指導教授 張貿翔 簽章

100 年 6 月 7 日

國立中正大學博士學位論文考試審定書

資訊工程學系

研究生林莊傑 所提之論文

A Study on Fixed-Parameter Algorithms and Property Testing

經本委員會審查，符合博士學位論文標準。

學位考試委員會召集人 王有禮 簽章

委員

楊昌彪 P. Rossmanith

吳邦一 張貿翔

王炳豐 [illegible]

趙坤茂 [illegible]

指導教授 P. Rossmanith 張貿翔 簽章

中華民國 100 年 7 月 26 日

A Study on Fixed-Parameter Algorithms and Property Testing

Chuang-Chieh Lin

Principle Supervisor: Professor Maw-Shang Chang

Co-Supervisor: Professor Peter Rossmanith

Dissertation Submitted To
Department Of Computer Science And Information Engineering
In Partial Fulfillment Of The Requirements
For The Degree Of

Doctor Of Philosophy

At

National Chung Cheng University
168, University Rd., Min-Hsiung Chia-Yi, Taiwan, R.O.C.

July, 2011

Dedicated to

my dear wife, Maggie Shu-Chun Kuo,

and

my lovely daughter, Sherry Liang-Yu Lin.

Acknowledgments

First and foremost I would like to give my best regards to my principle advisor, Professor Maw-Shang Chang (張貿翔). I appreciate his guide and advices for everything about doing research, and all kinds of supports during this period of time. I have greatly benefited and enjoyed from his courses and our meetings. He is not only a greatest theoretical computer scientist, but also a very kind and patient person.

I am very grateful to my co-advisor, Professor Peter Rossmanith, for indispensable discussions, stimulating ideas, and encouragement in my research and writing of this dissertation. His expertise on fixed-parameter algorithms and randomized algorithms guides me to accomplish this dissertation. Moreover, he also taught me how to work hard and enjoy life as well. I am at a loss for words.

Thank my master thesis supervisor, Professor Richard Chia-Tung Lee (李家同), for introducing me the field of algorithms and computational biology. Due to his encouragement, I started the journey of pursuing a Ph.D. degree in theoretical computer science.

I am deeply indebted to my parents Chen-Pao Lin (林鎮寶) and Ming-Chu Du (杜明珠). They have toiled extremely hard to bring me up and educate me. I have much gratitude for their love and support.

I would like to thank the dissertation committee members, Professor Chang-Biau Yang (楊昌彪), Professor Yao-Ting Huang (黃耀廷), Professor Yue-Li Wang (王有禮), Professor Kun-Mao Chao (趙坤茂), Dr. Chi-Jen Lu (呂及人), Professor Biing-Feng Wang (王炳豐), Professor Bang-Ye Wu (吳邦一), Professor Peter Rossmanith and Professor Maw-Shang Chang. Their comments and questions make the dissertation more likely to be thoughtful and complete.

The Computation Theory group of the department has been a source of friendships as well as good advices and collaborations. I would like to acknowledge my two colleagues, Dr. Wu-Hsiung Wu (吳武雄) and Ms. Ling-Ju Hung (洪綾珠), for creating a pleasant working atmosphere and providing me all kinds of assistance. Thank Ms. Ling-Ju Hung for arranging everything for my dissertation defense oral examination. Especially, I want to thank Professor Chuan-Min Lee (李權明), who was a former member of the group. As his research assistant, I appreciate the opportunity to work with him.

I would like to thank my friends in Aachen. During the days of study in Germany for the DAAD-NSC Sandwich Program, Ms. Birgit Willms provided me with all kinds of assistance for many life affairs. Thank the members of the Theoretical Computer Science group in RWTH Aachen University, especially Mr. Alexander Langer and Dr. Joachim Kneis, for their helpful discussions and opinions. The wonderful lunch time with them in Peking Town is unforgettable for me. I want to thank Josef Kunze, who is one of my best friends, for saving my teeth many times and taking care of me when I was seriously ill.

I want to thank Dr. Antonius Jacobus Johannes Kloks (Ton Kloks) for many useful discussions on graph theory and combinatorial mathematics. His expertise not only helps me solve numerous mathematical problems, but also inspires me to come up with ideas of some proofs in the dissertation. I thank Dr. Mark Chun-Jung Su (蘇俊榮), who is also one of my best friends, for always sharing important information with me. Without his help, I might have got into trouble many times.

Finally, I would like to show my special thanks to my wife, Maggie Shu-Chun Kuo (郭淑君). As a husband, I am indescribably blessed by her love and support. I am so lucky that I have her. In addition, I thank my lovely daughter, Sherry Liang-Yu Lin (林亮予), for giving the mouse and the keyboard back to me, so that I can complete this dissertation. This dissertation is dedicated to them, who are my most beloved. I love them forever and ever.

Abstract

For a long time, as long as a problem is proved to be **NP**-hard, people usually avoid solving it exactly due to its computational hardness. In fact, there are strategies of designing *fixed-parameter algorithms*, which can be used to solve these problems exactly. A parameterized problem is a language $L \subset \Sigma^* \times \mathbb{Z}^+$, where Σ is a finite alphabet. The first component is called the problem instance of L which has size of n, and the second component, which is simply a nonnegative integer k for most cases, is called the parameter of L. A *fixed-parameter algorithm* is an algorithm that solves a parameterized problem in $f(k) \cdot n^{O(1)}$ time for some computable function f depending solely on k. When k is small, a fixed-parameter algorithms runs in poly(n) time. In the past two decades, a variety of useful methods and techniques for demonstrating fixed-parameter tractability or designing fixed-parameter algorithms have emerged.

Besides, with recent advances in technology, we are faced with imperious need to process increasing larger amounts of data quickly. It is sometimes necessary to come out an answer without examining the whole input, yet the answer must have guaranteed accuracy. *Property testing* delves into the possibilities of getting answers by observing only a small fraction of the input. An input, given as a function $f : D \mapsto F$, is said to be ϵ-*close* to satisfying a property $\mathcal{P}$, if there exists a function $f' : D \mapsto F$ that satisfies $\mathcal{P}$ and differs from f in less than $\epsilon|D|$ places. Otherwise, it is said to be ϵ-*far* from $\mathcal{P}$. Given a specified property $\mathcal{P}$, property testing is the study of the following task: *Given queries or accesses to an unknown function f, determine in $o(|D|)$ time whether f satisfies $\mathcal{P}$ or is ϵ-far from $\mathcal{P}$.* In the past decade, property testing has become one of the most active fields in theoretical computer science.

In this dissertation, we study fixed-parameter algorithms and property testing, and introduce a new concept: *parameterized property testing*, which combines the

characteristics of these two fields. Given a function $f : D \mapsto F$, $\epsilon \in (0,1)$, and an integer $k \in \mathbb{Z}^+$ as the parameter, a *parameterized property tester* for a property $\mathcal{P}$ is a property tester for $\mathcal{P}$ which has time complexity $\phi(k, 1/\epsilon) \cdot o(|D|)$, where ϕ is a function that solely depends on k and ϵ. In the first half of the dissertation, we focus on a problem of determining consistency of a set of quartet topologies, which is related to evolutionary tree reconstruction. We tackle this problem and its variants through the aspects of fixed-parameter algorithms, property testing and parameterized property testing. Let Q be a set of quartet topologies over an n-taxon set S. We say that Q is *complete* if every quartet over S has exactly one topology in Q. Given a complete Q, the *Minimum Quartet Inconsistency* (MQI) problem asks if there exists an unrooted evolutionary tree T such that at most k quartet topologies in Q are not satisfied by T. For the MQI problem, we present three fixed-parameter algorithms with time complexity $O(3.0446^k n + n^4)$, $O(2.0162^k n^3 + n^5)$, and $O^*((1+\varepsilon)^k)$, respectively, where $\varepsilon > 0$ is an arbitrarily small constant. Next, we consider *tree-consistency* of quartet topologies, which is the property that all the quartet topologies in Q are satisfied by an unrooted evolutionary tree. To test if a complete Q is tree-consistent, we give a non-adaptive $O(n^3/\epsilon)$ property tester with one-sided error. When Q is not necessarily complete, we give a non-adaptive $O(1.7321^k k n^3/\epsilon)$ parameterized property tester with one-sided error to test if Q is tree-consistent, where $k \in \mathbb{Z}^+$ is an upper bound on the number of quartets which do not have topologies in Q. This parameterized property tester is uniform on k.

In the second half of the dissertation, we study parameterized property testing for graph properties and focus on two **NP**-hard graph theoretical problems: the *Vertex Cover* problem and the problem of computing *treewidth* of a graph. We consider the sparse model, where graphs are stored in adjacency lists and have maximum vertex degree bounded by d. To test if an n-vertex graph has a vertex cover of size at most k, we present an adaptive parameterized property tester with two-sided error, which runs in $O(d/\epsilon)$ time for $k < n/(6d)$, and another adaptive parameterized property tester with one-sided error, which runs in $O(kd/\epsilon)$ time for $k < \epsilon n/4$. For testing if an n-vertex graph has treewidth at most k, we give two adaptive parameterized property testers with two-sided error, which run in $2^{d^{O(kd^3/\epsilon^2)}}$ time and $d^{(k/\epsilon)^{O(k^2)}} + 2^{\mathrm{poly}(k,d,1/\epsilon)}$ time respectively. Both of them are uniform on k.

摘要

長久以來, 當一個問題被證明爲 **NP**-hard 之後, 因爲計算複雜度高的緣故, 人們總是避免去直接地去求這個問題的最佳解。事實上, 有一些設計固定參數演算法的策略, 讓我們可以用來直接去解這些過去避而不談的問題。一個參數化問題爲一語言 $L \subset \Sigma^* \times \mathbb{Z}^+$, 其中 Σ 爲一個有限的字母集。L 的第一個部份爲大小爲 n 的問題實例, 而第二個部份爲一個非負整數 k, 稱爲 L 之參數。一個能在 $f(k) \cdot n^{O(1)}$ 的時間複雜度內解決一個參數化問題的演算法, 我們稱之爲固定參數演算法, 其中 f 是一個只跟 k 相依的可計算函數。當 k 之值很小時, 固定參數演算法能在 n 的多項式時間內執行完畢。在過去二十年來, 證明一個問題存在固定參數演算法和設計各種不同固定參數演算法的方法與技巧, 不斷地被開發出來。

此外, 隨著科技不斷地進步, 我們亟需更快速地處理大量資料。有時候, 我們必須在不看完全部輸入資料的要求下得到答案, 並且仍確保答案的正確性。性質測試探討只看輸入資料的一小部份而能得到答案的可能性。給定一個函數 $f : D \mapsto F$ 爲輸入, 如果存在一個函數 $f' : D \mapsto F$ 滿足某一個性質 $\mathcal{P}$ 而且 f' 與 f 對應的函數值不同之處少於 $\epsilon|D|$ 個位置, 我們稱 f 爲 ϵ-接近於性質 $\mathcal{P}$, 否則, 我們稱 f 爲 ϵ-遠離於性質 $\mathcal{P}$。給定一個性質 $\mathcal{P}$, 性質測試最主要的工作如下: 透過查詢或存取一個未知的函數 f, 在 $o(|D|)$ 的時間複雜度內判斷 f 是否滿足性質 $\mathcal{P}$, 抑或 ϵ-遠離於性質 $\mathcal{P}$。在過去十年來, 性質測試已經成爲理論計算機科學中最熱門的領域之一。

在本論文中, 我們針對固定參數演算法與性質測試進行研究, 並結合了這兩者的的特性進而提出一個新的概念:「參數化性質測試」。給定一個函數 $f : D \mapsto F$, $\epsilon \in (0, 1)$, 以及一個整數 $k \in \mathbb{Z}^+$ 作爲參數, 針對一個性質 $\mathcal{P}$ 的參數化性質測試演算法即爲一個時間複雜度爲 $\phi(k, 1/\epsilon) \cdot o(|D|)$ 的性質測試演算法, 其中 ϕ 爲一個只與 k 和 ϵ 相依的函數。在本論文的前半段, 我們聚焦在一個與演算樹重建相關的問題: 決定一組四元拓樸集的一致性。我們利用固定參數演算法、性質測試與參數化性質測試這三個方向去處理這個問題與其變形。令 Q 爲一組在 S 上的四元拓樸集, 其中 S 爲一個包含 n 個物種的集合。若每個 S 上的四元集在 Q 中都恰好存在一個對應的拓樸, 我們稱 Q 爲完整的。給定一組在 S 上的完整四元拓樸集, 最少四元樹不一致問題是問「是否存在一個無根的演化樹 T, 使得 Q 中至多 k 四元拓樸無法被

T 滿足」。針對最少四元樹不一致問題, 我們提出三個固定參數演算法, 其時間複雜度分別為 $O(3.0446^k n + n^4)$, $O(2.0162^k n^3 + n^5)$, 與 $O^*((1+\varepsilon)^k)$, 這裡的 ε 是一個任意小的正值常數。接著, 我們考慮四元拓樸集的「樹一致性」。若存在一個無根演化樹滿足在 Q 中的所有四元拓樸, 我們稱 Q 具有樹一致性。針對一個完整的四元拓樸集 Q, 我們提出一個 $O(n^3/\epsilon)$ 性質測試演算法來測試 Q 是否具有樹一致性, 該演算法具有非遷就性與單邊誤差。當 Q 未必完整時, 我們提出一個時間複雜度為 $O(1.7321^k k n^3/\epsilon)$ 參數化性質測試演算法來測試 Q 是否具有樹一致性, 其中 $k \in \mathbb{Z}^+$ 為在 Q 中不具有拓樸之四元集的數目上限。此參數化性質測試演算法同樣具有非遷就性與單邊誤差, 且在 k 上均勻一致。

在本論文的後半段, 我們探討圖形性質的參數化性質測試, 並聚焦在圖形理論上的兩個 **NP**-hard 問題:「點覆蓋問題」與「樹寬計算問題」。我們考慮稀疏圖模型, 在此模型下的圖形儲存於相鄰串列裡, 且每個點至多有 d 個相鄰點。針對測試一個 n-點圖形是否具有大小至多為 k 的點覆蓋集, 我們在稀疏圖模型下, 提出兩個具遷就性的參數化性質測試演算法, 第一個演算法具有雙邊誤差, 在 $k < n/(6d)$ 的時候, 其時間複雜度為 $O(d/\epsilon)$。第二個演算法則具有單邊誤差, 在 $k < \epsilon n/4$ 的時候, 其時間複雜度為 $O(kd/\epsilon)$。針對測試一個 n-點圖形之樹寬是否大小至多為 k, 我們在稀疏圖模型下提出兩個具遷就性與雙邊誤差的參數化性質測試演算法, 其時間複雜度分別為 $2^{d^{O(kd^3/\epsilon^2)}}$ 與 $d^{(k/\epsilon)^{O(k^2)}} + 2^{\mathrm{poly}(k,d,1/\epsilon)}$。這兩個演算法在 k 上皆為均勻一致。

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Fixed-Parameter Algorithms

The monograph by Gary and Johnson [72] provides a comprehensive and thorough study of **NP**-completeness, which implies computational intractability of many problems. For a long time, to cope with intractable problems, people have referred to approximation algorithms or purely heuristic methods.

In fact, we usually find that an **NP**-hard problem is easy to deal with if some parameter of the problem instance is small. To design algorithms for such problems leads to the notion of *fixed-parameter algorithms* . A *parameterized problem* is a language $L \subset \Sigma^* \times \Sigma^*$, where Σ is a finite alphabet. The first component of L is the problem instance, and the second component of L is called the parameter. In most cases, the parameter is a nonnegative integer which is denoted by k. Generally speaking, a fixed-parameter algorithm is an algorithm that determines whether $(x, k) \in L$ (i.e., solves the parameterized problem) in $f(k) \cdot n^{O(1)}$ time for some computable function f solely depending on k, where $n = |x|$. Such algorithms then bring out a class of problems called *fixed-parameter tractable* (**FPT**), in which a problem admits a fixed-parameter algorithm. Obviously, a fixed-parameter algorithm runs in polynomial time when $f(k)$ is regarded as a constant. Note that an algorithm with running time $O(n^{f(k)})$ is not our concern, since it is much slower.

To make readers grasp the rough idea of fixed-parameter algorithms quickly, we use the *Vertex Cover* problem as an illustrating example, which is defined below. Readers who are unfamiliar with the fundamental notions of graphs are suggested to refer to Appendix A.

The Vertex Cover problem
Input: A graph $G = (V, E)$ and an integer $k \geq 0$.
Task: Determine if there exists a vertex subset $C \subseteq V$ of size at most k such that each edge in E has at least one of its endpoints in C.

Let $G = (V, E)$ be the input graph. A vertex subset $C \subseteq V$ is called a *vertex cover* if each edge in the graph has at least one of its endpoints in C. The Vertex Cover problem is to determine whether there exists a vertex cover of size at most k for the input graph G. The Vertex Cover problem is a well-known **NP**-complete problem [72], thus it seems hopeless to devise an efficient algorithm for this problem. However, let us consider the following observation. For $C \subseteq V$ to be a feasible solution to the Vertex Cover problem, each edge $(u, v) \in E$ in the graph must be *covered* by C, that is, at least one of its endpoints must be in C. Based on this observation, we pick one of $\{u, v\}$, say u, into the solution and delete u together with its incident edges, and then continue recursively with the remaining graph. Such a recursive algorithm works as a search tree where each tree-node corresponds to a certain recursion and has two branches. The depth of the search tree is bounded by k. Let $T(k)$ denote the number of leaves of the search tree, then we have $T(k) = T(k-1) + T(k-1)$. Since the search tree is binary, it is clearly that $T(k) \leq 2^k$. Note that it takes $O(n)$ time for each tree-node (i.e., the time cost for deleting the incident edges of a vertex). Thus, the overall time complexity of the algorithm is clearly $O(2^k n)$. Such an algorithm is efficient when the parameter k is small. This example suggests the possibility of designing efficient algorithms to solve the Vertex Cover problem exactly for small k's.

In fact, the search tree size can be even smaller. Let us consider another recursive algorithm: Simple-VC, whose pseudocode is listed in Algorithm 1.1. The variables $v_{\max}$, `sum_degs`, and `max_deg` denote the vertex with maximum vertex degree, the sum of vertices degrees, and the maximum vertex degree in the graph, respectively. Assume that $v_{\max}$, `sum_degs`, and `max_deg` are initialized to be $\varnothing$, 0 and 0 respectively. We clarify the general idea of the algorithm as follows. First, we remove the isolated vertices (i.e., the vertices with degree 0) in the graph. Note that if a vertex cover C contains isolated vertices, then removing these isolated vertices from C still results in a vertex cover since an isolated vertex does not cover any edge. Second, we find out the maximum degree (i.e., `max_deg`) and a vertex with the maximum degree (i.e., $v_{\max}$) in the graph. If the maximum degree of the graph is 1, then it is clear that the graph G consists of disjoint edges (i.e., edges that mutually share no endpoint). For this case, the size of any vertex cover of G is equal to the number of edges in G, which is equal to half of `sum_degs`. Then the algorithm answers "yes" or "no" by comparing k with the number `sum_degs`/2. Otherwise, consider the vertex $v_{\max}$. In order to cover the incident edges of $v_{\max}$, either $v_{\max}$ or its neighbors (i.e., $N_G(v_{\max})$) must be selected into the vertex cover. The algorithm recursively branches on these two cases. For the former (i.e., the algorithm selects $v_{\max}$ into the vertex cover), it removes $v_{\max}$ and the incident edges of $v_{\max}$ from the graph, and then decreases the value of k by 1. For the latter (i.e., the algorithm selects $N_G(v_{\max})$ into the vertex cover), it removes $N_G[v_{\max}]$ (i.e., $\{v_{\max}\} \cup N_G(v_{\max})$) and their incident edges from the graph, and then decreases the value of k by the size of $N_G(v_{\max})$.

Algorithm Simple-VC takes $O(n+m)$ time for computing $v_{\max}$, `max_deg`, `sum_degs`, and the remaining graphs, say G_1 and G_2, respectively. The recursion of the algorithm also works as a search tree with depth bounded by k. Let T denote the number of leaves of the search tree. Note that the parameter k is decreased by at least two at Line 20 since the maximum degree of the graph is greater than one due to the processes that were done in prior. Thus, we derive that $T(k)$ is bounded by $T(k-1) + T(k-2)$. At the first sight, we can only obtain $T(k) \leq T(k-1) + T(k-2) \leq T(k-1) + T(k-1) \leq 2^k$. However, by the approach introduced in Sect. 2.2, we can obtain that $T(k-1) + T(k-2) \leq 1.62^k$, which leads to $T(k) \leq 1.62^k$. Thus this algorithm solves the Vertex Cover problem in $O(1.62^k(m+n)) = O(1.62^k n^2)$ time.

The above examples of solving the Vertex Cover problem reveal the possibility of deriving fixed-parameter algorithms whose time complexity has much less exponential dependency on k. This makes such algorithms efficient in practical uses when k is small. Such a parameter k is relevant to the "target" of an **NP**-hard optimization problem.

Generally, there are also parameters which are relevant to the "structure" of the problem instance. The *treewidth* of a graph is a typical example for this case.

```
Simple-VC(G, k) /* a graph G = (V, E) and an integer k as the parameter */
begin
for each v ∈ V do
   sum_degs ← sum_degs + deg_G(v);
   if deg_G(v) = 0 then
      G ← G − {v};
   else if max_deg < deg_G(v) then
      v_max ← v;
      max_deg ← deg_G(v);
   end if
end for
if max_deg ≤ 1 then
   if sum_degs/2 ≤ k then
      return "yes";
   else
      return "no";
   end if
else
   G_1 ← G − {v_max};
   G_2 ← G − ({v_max} ∪ N_G(v_max));
   Simple-VC(G_1, k−1); /* v_max is selected into the vertex cover */
   Simple-VC(G_2, k−|N_G(v_max)|); /* N_G(v_max) is selected into the vertex cover */
end if
end
```

Algorithm 1.1: Simple-VC: a simple $O(1.62^k(n+m))$ fixed-parameter algorithm for the Vertex Cover problem.

Roughly speaking, the treewidth of a graph measures "how close a graph is to being a tree" (refer to Sect. 5.2 for the formal definition and more details). It is one of the most fundamental notions in graph theory and algorithms. Given a graph G, the treewidth of G can be derived by computing the *tree-decomposition* of G which can be done in $2^{\Theta(k)} \cdot k^{O(1)} \cdot n$ time [28]. With the tree-decomposition of a graph at hand, many **NP**-hard problems, such as the Vertex Cover problem, the Maximum Independent Set problem, the Minimum Dominating Set problem, the Hamiltonian Cycle problem, the problem of computing the chromatic number of a graph, etc., can be solved in $O(n)$ time when the treewidth of the input graph is bounded by a fixed k [14, 98]. Furthermore, Courcelle [52] proved that graph theoretical problems that can be formulated as monadic second-order logic (MSO) formulae are $O(n)$-time solvable when the treewidth is bounded by a fixed k.

The above examples implicitly reveals the fact that some **NP**-hard problems are difficult only when the parameters get large. Generally speaking, the main tasks in the field of parameterized complexity theory include finding the parameters which make **NP**-hard problems difficult and improving currently known **FPT** results. Take the Vertex Cover problem as an example. The size of a vertex cover is a kind of parameter which makes the Vertex Cover problem difficult. The problem becomes more difficult when the size of the minimum vertex cover of the input graph gets larger. As to the algorithmic improvement of solving this problem, we have seen a simple $O(2^k n)$ fixed-parameter algorithm, and an $O(1.62^k n^2)$ fixed-parameter algorithm. The current best fixed-parameter algorithm runs in $O(1.2738^k + kn)$ time [48]. The base of the exponential function of k decreases significantly.

Although there has been many **NP**-hard problems shown to be fixed-parameter tractable, there exist **NP**-hard problems that do not admit fixed-parameter algorithms unless $\mathbf{NP} = \mathbf{P}$. For example, let us consider the Independent Set problem. Given a graph $G = (V, E)$, a vertex subset $S \subseteq V$ is an *independent set* if none of the pairs of vertices in S are adjacent. The Independent Set problem asks if there is an independent set of size k in the graph. It is well-known that G has a vertex cover of size k if and only if it has an independent set of size $n - k$. Let k' denote $n - k$. If the Independent Set problem with the parameter k' is fixed-parameter tractable, then the Vertex Cover problem with the parameter k can be solved efficiently even though k is quite large. From this point of view, we can realize that the Independent Set problem is unlikely to be fixed-parameter tractable. In fact, there are problems shown to be *fixed-parameter intractable* and the hierarchy with respect to their difficulty has been established. See [60] for more details.

In the past two decades, fixed-parameter algorithms have been extensively studied. There are excellent surveys and textbooks introducing this field. For instance, the work by Downey and Fellows [60] in 1999 is one of the best monographs for introducing fixed-parameter algorithms and the parameterized complexity. Later in 2006, Niedermeier [98] wrote an elaborate textbook for introducing fixed-parameter algorithms and parameterized complexity. Many approaches for designing fixed-parameter algorithms are summarized in this book. Readers are recommended to refer to the above literatures for more information.

1.2 Property Testing

By observing recent advances in technology of the real world, we are faced with imperious need to process increasing larger amounts of data quickly. Many practical problems have inputs of very large size, so that even taking a linear time in its size to provide an answer is too much. It is sometimes necessary to come out an answer quickly without examining the whole input, yet the answer must have guaranteed accuracy. *Property testing* is a new field in computational complexity theory and algorithm design. It delves into the possibilities of getting answers (*yes* or *no* for decision problems) by observing only a small fraction of the input. The notion of property testing provides an aspect that how a decision problem can be "approximated". To achieve the goal of property testing, randomized algorithms are always used, however, the probability of getting an erroneous answer should be very small.

Let us clarify the general concepts of property testing by considering functions as follows. Let $\mathcal{F}$ be the set of all functions with the same domain D. Let $\mathcal{P}$ be a fixed property of functions in $\mathcal{F}$, which can be viewed as a subset of $\mathcal{F}$. For two functions f and g in $\mathcal{F}$, let $\delta(f,g)$ denote the fraction of the points in the domain D where f and g have different values. Obviously, the range of δ is $[0,1]$. Then for a function $f \in \mathcal{F}$, we define that $\Delta(f,\mathcal{P}) = \min_{g\in\mathcal{P}} \delta(f,g)$. We say that f *satisfies the property* $\mathcal{P}$ if $\Delta(f,\mathcal{P}) = 0$. We say f is ϵ*-far* from $\mathcal{P}$ if $\Delta(f,\mathcal{P}) \geq \epsilon$, otherwise f is said to be ϵ*-close* to satisfying $\mathcal{P}$. According to the above notations, a *property tester* for $\mathcal{P}$ is defined as follows.

Definition 1.1 (Property testers [75])**.** Given a function $f \in \mathcal{F}$ and a parameter $0 < \epsilon < 1$ as the input, a property tester for $\mathcal{P}$ is an algorithm $\mathcal{M}$ such that the following conditions hold:

1. $\mathcal{M}$ runs in $o(|D|)$ time;
2. $\mathcal{M}$ returns "yes" with probability at least 2/3 if $\Delta(f,\mathcal{P}) = 0$ (i.e., $f \in \mathcal{P}$);
3. $\mathcal{M}$ returns "no" with probability at least 2/3 if $\Delta(f,\mathcal{P}) \geq \epsilon$.

Moreover, a property tester $\mathcal{M}$ for property $\mathcal{P}$ is said to have *one-sided error* if it returns "yes" for every instance satisfying $\mathcal{P}$ with probability 1. If $\mathcal{M}$ makes queries without knowing the results of previous ones, we say that $\mathcal{M}$ is *non-adaptive*. A property $\mathcal{P}$ is called *testable* if it has a property tester that runs in $q(\epsilon)$ time, where $q(\epsilon)$ is independent of the input size. Moreover, we say that $\mathcal{P}$ is *easily testable* if it has a property tester which has one-sided error and runs in $\text{poly}(1/\epsilon)$ time. Note that $o(\cdot)$ in the first condition of Definition 1.1 is an asymptotic notation of "little-o". For functions $f, g : \mathbb{Z}^+ \mapsto \mathbb{R}^+$, where $\mathbb{Z}^+$ and $\mathbb{R}^+$ denote the set of nonnegative integers and the set of nonnegative real numbers, respectively, we denote by $f(x) = o(g(x))$ if $\lim_{x\to\infty} f(x)/g(x) = 0$.

Generally, from the point of view of algorithm design, one prefers non-adaptive testers to adaptive ones due to the reason that the strategy of sampling all at once suffices. When time complexity is the most concern, the result that a property is easily testable is better than that it is testable. Be noted that the time complexity of any property tester should be sublinear in the domain size. From the point of view of correctness, a tester with one-sided error tester is better than another one with two-sided error. For any property, a non-adaptive property tester usually requires more time than an adaptive one, and a property tester with one-sided error usually requires more time than that with two-sided error.

Here let us consider testing emptiness of a graph as an illustrating example of property testing. We say that a graph $G = (V, E)$ satisfies *emptiness* if $E = \emptyset$, that is, there exists no edge in the graph G. Suppose that the *dense model* of graphs is applied. In the dense model, a graph G is represented by an adjacency matrix where an algorithm is allowed to make *queries*. Here a query means to examine whether an entry of the matrix equals to 1 or 0, that is, to see if a pair of vertices are adjacent or not. The distance measure of two graphs refers to the fraction of vertex pairs which is an edge in one graph and not an edge in the other, taken over the domain size which is n^2. Hence, G is ϵ-far from emptiness if it has at least ϵn^2 edges. Let us consider a randomized algorithm, which is called Emptiness-Tester, in Algorithm 1.2.

Algorithm Emptiness-Tester first picks $2/\epsilon$ vertex pairs uniformly at random, and then check if any of them is a pair of adjacent vertices. Once a pair of adjacent vertices is found, the algorithm returns "no" since it finds an evidence that the graph is not empty. If none of them is a pair of adjacent vertices, then the algorithm returns

```
Emptiness-Tester(G) /* a graph G = (V, E) stored in an adjacency matrix */
begin
  pick 2/ε vertex pairs from G uniformly at random;
  for each picked vertex pair (u, v) do
     if (u, v) ∈ E then
        return "no";
     end if
  end for
  return "yes";
end
```

Algorithm 1.2: Emptiness-Tester: a property tester for testing emptiness of a graph in the dense model.

"yes". It is easy to see that if G is really empty, then there is no edge in G so that the algorithm must return "yes". On the other hand, if the graph is ϵ-far from being empty, then there must be at least ϵn^2 pairs of vertices that are adjacent. In this case, the algorithm returns "no" with probability at least

$$1 - \left(1 - \frac{\epsilon n^2}{n^2}\right)^{2/\epsilon} = 1 - ((1-\epsilon)^{1/(-\epsilon)})^{-2} > 1 - e^{-2} > \frac{2}{3}.$$

Algorithm Emptiness-Tester utilizes only $O(1/\epsilon) = o(n^2)$ queries, thus it is indeed a valid property tester for testing emptiness of a graph. Clearly, we obtain that emptiness of a graph is easily testable in the dense model.

The general notion of property testing was first explicitly formulated by Rubinfeld and Sudan [108], who were motivated by the connection to the *program checking* [25]. Suppose we have a program P which calculates a function f over a domain $\mathcal{D}$. A so-called *ϵ-self-tester* is to distinguish whether the program P truly calculates the function f or has wrong calculation results for more than $\epsilon|\mathcal{D}|$ points of the domain $\mathcal{D}$. Then the authors extend the self-testers to *ϵ-function-family testers*, which take the program P as an input and test if there exists a function $f \in \mathcal{F}$ such that P has wrong answers at less than $\epsilon|\mathcal{D}|$ points in the domain, where $\mathcal{F}$ is a certain family of functions possessing a certain property. The study on testing combinatorial objects was first introduced by Goldreich, Goldwasser, and Ron [75].

Recall that, in the dense model, a graph is stored in an adjacency matrix. A property tester is allowed to make *queries*, where each query is to examine an entry (i, j) in the adjacent matrix in order to know whether vertex i and j are adjacent or not. The input graph is ϵ-far from a property $\mathcal{P}$ if more than ϵn^2 edge insertions or removals should be performed to make the graph have the property. In [75], many graph properties, such as k-colorability, bipartiteness, having a large clique, having a large cut, etc., were proved to be testable in the dense model. In [3] it was shown that every first-order graph property without a quantifier alternation of type '$\forall\exists$' is testable. Later, *monotone graph properties* and *hereditary graph properties* are shown to be testable in the dense model [10, 11]. Monotone graph properties are the graph properties that are closed under removal of vertices and edges, while hereditary graph properties are the graph properties that are closed under vertex removals. A graph is *H-free* if it does not contain any subgraph isomorphic to H, and it is *induced H-free* if it does not contain any *induced* subgraph isomorphic to H. Clearly, induced H-freeness is a monotone graph property and induced H-freeness is a hereditary graph property. The property H-freeness is easily testable if H is bipartite [2]. In [9], Alon and Shapira gave a nearly complete characterizations of H's such that induced H-freeness is easily testable, though it is still open that whether induced P_4-freeness and induced C_4-freeness are easily testable, where P_4 is a path of length 3 and C_4 is a cycle of length four. Table 1.1 summarizes the results on testing graph properties in the dense model.

There is another frequently used graph model, which is called the *sparse model.* In this model, bounded-degree graphs are considered and stored in adjacency lists. Goldreich and Ron [76] are the first ones to study property testing in the sparse model. Unlike property testing in the dense model, there are only a few graph properties shown to be testable in the sparse model. Recently, there are breakthroughs of property testing in this model. In [21, 83], minor-closed properties are shown to be testable in the sparse model. When we focus on the testing for special classes of graphs, hereditary graph properties are proved to be testable when the input graph has very limited expansion [54]. Very recently, property of hyperfinite graphs are proved to be testable in the sparse model [21, 83]. Table 1.2 summarizes the results on testing graph properties in the sparse model.

There are also non-graph properties studied in the field of property testing, such as testing monotonically nondecreasing of a sequence of numbers [63], testing constraint satisfiability [7], testing whether a language is regular [5] (the results are

Property	Testable	Easily testable	Query complexity
first-order graph properties without a quantifier alternation of type '∀∃'	Yes [3]	No [75]	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}O(\mathrm{poly}(1/\epsilon))$ 2's [3]
first-order graph properties with a quantifier alternation of type '∀∃'	No [3]	No [3]	$\star$
monotone properties	Yes [10]	No [2]	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}O(\mathrm{poly}(1/\epsilon))$ 2's [10]
hereditary properties	Yes [11]	No [9]	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}O(\mathrm{poly}(1/\epsilon))$ 2's [11]
H-freeness, H is bipartite	Yes [3]	Yes [2]	$O(h^2\left(\frac{1}{2\epsilon}\right)^{h^2/4})$ [2]
H-freeness, H is not bipartite	Yes [3]	No [2]	$\Omega\left(\left(\frac{c}{\epsilon}\right)^{c\log(c/\epsilon)}\right)$ [2]
induced H-freeness, $H = P_2$	Yes [3]	Yes [9]	$\Theta\left(\frac{1}{\epsilon}\right)$
induced H-freeness, $H = P_3$	Yes [3]	Yes [9]	$O\left(\frac{\log(1/\epsilon)}{\epsilon}\right)$ [9]
induced H-freeness, $H \neq P_2, P_3, P_4, C_4$ or their complements.	Yes [3]	No [9]	$\Omega\left(\left(\frac{1}{\epsilon}\right)^{c\log(1/\epsilon)}\right)$
induced H-freeness, H is P_4 or C_4	Yes [3]	?	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}O(\mathrm{poly}(1/\epsilon))$ 2's [11]
bipartiteness	Yes [75]	Yes [75]	$O\left(\frac{\ln^8(1/\epsilon)\,\mathrm{lnln}^2(1/\epsilon)}{\epsilon^2}\right)$ [4]
k-colorability	Yes [75]	Yes [75]	$O\left(\frac{k^2\ln^2 k}{\epsilon^4}\right)$ [4]
having a clique of size at least ρn	Yes [75]	No [75]	$O\left(\frac{\log^2(1/\epsilon)\rho^2}{\epsilon^6}\right)$ [75]
having a cut of size at least ρn^2	Yes [75]	No [75]	$O\left(\frac{\log^2(1/\epsilon)}{\epsilon^7}\right)$ [75]

Table 1.1: Important results on testing graph properties in the dense model. $h = |H|$; c is a constant depending on H; '?' stands for an open question; '$\star$' means no explicit bound is given.

Property	Testable	Easily testable	Query complexity
properties of hyperfinite graphs	Yes [95]	?	$\star$ [95]
hereditary properties in a nonexpanding family of graphs	Yes [54]	?	$\star$ [54]
minor-closed properties	Yes [21]	?	$2^{\text{poly}(1/\epsilon)}$ [83]
bipartiteness	No [76]	No [76]	$\Omega(\sqrt{n})$ [76]
expansion	No [76]	No [76]	$\Omega(\sqrt{n})$ [76]
k-colorability	No [33]	No [33]	$\Omega(n)$ [33]
connectivity	Yes [76]	Yes [76]	$O\left(\frac{\log^2(1/\epsilon d)}{\epsilon}\right)$ [76]
k-connectivity	Yes [118]	Yes [118]	$O\left(d\left(\frac{ck}{\epsilon d}\right)^k \log\frac{k}{\epsilon d}\right)$ [118]
k-edge-connectivity for $k = 1, 2$	Yes [76]	Yes [76]	$O\left(\frac{\log^2(1/\epsilon d)}{\epsilon}\right)$ [76]
k-edge-connectivity for $k \geq 4$	Yes [76]	Yes [76]	$O\left(\frac{k^3 \log(1/(\epsilon d))}{\epsilon^{3-2/k} d^{2-2/k}}\right)$ [76]
Eulerian	Yes [76]	Yes [76]	$O\left(\frac{\log^2(1/\epsilon d)}{\epsilon}\right)$ [76]
cycle-freeness	Yes [76]	No [76]	$O\left(\frac{1}{\epsilon^3}\right)$ [76]*

Table 1.2: Important results on testing graph properties in the sparse model. '$\star$' stands for a bound in a not explicitly form yet it is independent of n; '?' stands for an open question; '*' stands for a result with two-sided error.

then extended to the testing on read-once branching programs [92] and read-twice branching programs [70], which are testable and non-testable respectively), etc. In [18], Batu et al. considered testing whether two distributions of n elements are closed. Property testing also emerges in the context of probabilistically checkable proof (**PCP**) systems [15, 58], and a variant of the **PCP** system similar to the setting of property testing is also studied [64]. Naturally, property testing is related to the notion of additive approximation [12, 69, 103, 104]. For more details of graph property testing, refer to [8, 67, 74, 106] for more details.

1.3 A New Concept: Parameterized Property Testing

As mentioned in [106], property testing may be useful in some scenarios. For example, suppose we have a slow exact decision procedure and a property tester for a function. If the property tester answers "no", then we know that with high probability the function does not have the property. In particular, for one-sided-error property testers, such a negative answer provides a witness that the function does not have the property, and therefore it is not necessary to run the slow decision procedure. Property testing is also useful when we can tolerate a small number of errors of the function values. In such a scenario, we only care whether the function is "good" (i.e., has the property) or "very bad" (i.e., ϵ-far from having the property). We have seen the examples of problems that can be efficiently solved when the associated parameters are small. One might be eager to know quickly whether the associated parameter of the problem is small or large so that the efficiency of the fixed-parameter algorithms can be expected to some degree. In such a scenario, using property testing as a preprocessing step helps.

On the other hand, the notion of fixed-parameter algorithms might also help property testing from the following point of view. Fixed-parameter algorithms are efficient when the associated parameters are small. Similarly, one might wonder whether a property tester can be more efficient when some associated parameter is small, or whether it facilitates the study of standard property testing. Based on the above idea, we introduce a new concept: *parameterized property testing*. For a specified property, it concerns whether there exists a property tester, which is called a *parameterized property tester*, such that the property can be tested efficiently when the parameter k associated with the input or the property is small. Based on this concept, we define parameterized property testers as follows.

Definition 1.2 (Parameterized property testers)**.** Given a function $f \in \mathcal{F}$, $0 < \epsilon < 1$, and $k \in \mathbb{Z}^+$ as the input, where $\mathcal{F}$ is the set of all functions with the same domain D, a *parameterized property tester* for a property $\mathcal{P}$ is an algorithm $\mathcal{M}$ such that the following conditions hold:

1. $\mathcal{M}$ runs in $O(\phi(k, 1/\epsilon) \cdot o|D|)$ time, where ϕ is a function which solely depends on k and ϵ;
2. $\mathcal{M}$ returns "yes" with probability at least 2/3 if $\Delta(f, \mathcal{P}) = 0$ (i.e., $f \in \mathcal{P}$);
3. $\mathcal{M}$ returns "no" with probability at least 2/3 if $\Delta(f, \mathcal{P}) \geq \epsilon$.

In Definition 1.2, we regard k (i.e., the parameter[1]) and ϵ as constants with respect to $|D|$. Hence, we say that a parameterized property tester runs in constant time if its time complexity solely depends on k and ϵ. A parameterized property tester $\mathcal{M}$ can be regarded as a collection of procedures $\mathcal{C}_{\mathcal{M}} = \{\Phi_k : k \in \mathbb{Z}^+\}$. We say that $\mathcal{M}$ is *nonuniform on k* if the procedures in $\mathcal{C}_{\mathcal{M}}$ are mutually distinct, otherwise we that it is *weakly uniform on k*. We say that $\mathcal{M}$ is *uniform on k* if the procedures in $\mathcal{C}_{\mathcal{M}}$ are all identical. If a property $\mathcal{P}$ with the parameter k admits a parameterized property tester of time complexity $\phi(k, 1/\epsilon)$ that solely depends on k and ϵ, we say that $\mathcal{P}$ is *parameterized testable*. If a parameterized testable property $\mathcal{P}$ admits an $O(\text{poly}(k, 1/\epsilon))$ parameterized property tester which is uniform on k and has one-sided error, then we say that $\mathcal{P}$ is *parameterized easily testable*.

In fact, there have been several examples of graph property testing implicitly revealing this idea. For example, Alon and Krivelevich [4] proposed an $O(k^2 \ln^2 k/\epsilon^4)$ property tester with one-sided error for k-colorability in the *dense model*. Their result implies that k-colorability is parameterized easily testable in the dense model. Alon [2] proved that testing if a graph is H-free (i.e., does not have H as a subgraph) requires $O(h^2(1/2\epsilon)^{h^2/4})$ queries in the dense model, where h is the size of $V(H)$. This result implies that H-freeness is parameterized testable in the dense model.

[1]In our setting of parameterized property testing, two or more parameters are allowed. For example, the maximum degree d in the sparse model for graph property testing is also regarded as a parameter.

As to the sparse model, where graphs of vertex-degree bounded by d are considered and adjacency lists are commonly used, Yoshida and Ito [118] obtained a property tester for k-connectivity, which runs in time $O(d(ck/\epsilon d)^k \log(k/\epsilon d))$ for some constant c. Yet for $k \geq \min\{\sqrt[3]{n/120}, \sqrt[3]{\epsilon dn/400}\}$, their property tester runs another $O(\mathrm{poly}(n)) = O(\mathrm{poly}(k/(\epsilon d)))$ algorithm to deterministically decide if the graph is k-connected. Hence, their property tester for k-connectivity in the sparse model is a parameterized property tester which is weakly uniform on k.

Note that there are property testing models called *massively parameterized models*, which generalize the setting of the standard property testing (e.g., see [68, 71, 82, 93, 94]). Yet, the parameter considered in a massively parameterized model is a fixed structure that determines all the input. For example, we can take K_n, which is a complete graph on n vertices, as the fixed structure. Then, the collection of inputs is the set of all 0/1 coloring of the edges of K_n, and the specified property is a subset of the collection of inputs. In this model, edge insertions and removals are forbidden. Except the massively parameterized models, to the best of our knowledge, we are not aware of any prior work that explicitly defined the same terminology as ours.

1.4 Our Contributions

In this dissertation, we introduce the new concept: parameterized property testing, which combines the characteristics of fixed-parameter algorithms and property testing. For the purpose of illustrating how to solve an **NP**-hard problem using fixed-parameter algorithms, property testing, and parameterized property testing, in the first part of the dissertation, we consider a problem of determining the *consistency of quartet topologies* as an example. The problem is about evolutionary tree reconstruction, which originates from computational biology. We tackle this problem and its variants through the approaches of fixed-parameter algorithms, property testing, and parameterized property testing.

First, we focus on the *parameterized Minimum Quartet Inconsistency* problem (parameterized MQI). Roughly speaking, a quartet topology is an unrooted tree with four leaves. Given a set Q of $\binom{n}{4}$ quartet topologies over an n-taxon set S, where each quartet over S has exactly one topology in Q, the parameterized MQI problem is to determine whether there exists an unrooted binary tree T, where internal nodes are of degree three and leaves are bijectively labelled by a set of n taxa, such that at most k quartet topologies (i.e., *quartet errors*) in Q are not consistent with T. Such an unrooted tree is called an *evolutionary tree*. In 2003, Gramm and Niedermeier showed that this problem is in **FPT** [78], and presented an $O(4^k n + n^4)$ fixed-parameter algorithm. We improve their result by devising three efficient fixed-parameter algorithms in a step-by-step way for the parameterized MQI problem. The complexity of these three algorithms are $O(3.0446^k n + n^4)$, $O(2.0162^k n^3 + n^5)$, and $O^*((1+\varepsilon)^k)$, respectively, where $\varepsilon > 0$ is an arbitrarily small constant[2]. Readers can also refer to [43] for this part of results.

Second, we consider property testing for the consistency of a set Q of quartet topologies. Our goal is to distinguish between the case that all the quartet topologies in Q are consistent with some evolutionary tree T and the case that no such evolutionary tree exists unless at least $\epsilon\binom{n}{4}$ quartet topologies in Q are changed. When there is exactly one topology in Q for every quartet over an n-taxon set S, we present an $O(n^3/\epsilon)$ property tester, which is non-adaptive and has one-sided error, for this property. This property tester is the first one for testing consistency of quartet topologies. Readers can also refer to [44] for this part of results.

Third, for the case that there are at most k quartets whose topologies are *missing* in Q (i.e., do not have topologies in Q), we show that there is an $O(1.7321^k kn^3/\epsilon)$ parameterized property tester, which is also non-adaptive and has one-sided error, for testing if Q is consistent with an evolutionary tree. Moreover, the parameterized property tester is uniform on k. Readers can refer to [45] for the preliminary result.

Finally, we study parameterized property testing for graph properties. We indicate that there are graph properties which are trivial to test (i.e., one can simply answer "yes" or "no" without observing the input graph) when the parameters are small constants, there are graph properties which are easily parameterized easily testable, and there are also graph properties which are not parameterized testable. Then, we focus on the Vertex Cover problem and the problem of computing treewidth of a graph, and give parameterized property testers for these properties in the sparse model, where graphs have bounded vertex degree d and are stored in adjacency lists.

[2]For two functions $f, g : (\mathbb{N}, \mathbb{N}) \to \mathbb{R}$, we write $f = O^*(g)$ if $f(n,k) = O(\text{poly}(n,k) \cdot g(n,k))$.

For testing whether a graph has a vertex cover of size at most k, Alon and Shapira's result [11] implies that there exists a property tester for this property in the dense model, yet its time complexity is only guaranteed to be a function of towers of 2's of height $O(\text{poly}(1/\epsilon))$. In the sparse model, it is proved in [76] that it requires at least $\Omega(\sqrt{n})$ time to test this property for $k = \rho n$, $\rho \in (0, 1)$. As to its parameterized complexity, the current best result for the Vertex Cover problem is $O(1.2738^k + kn)$ [48]. We present two adaptive parameterized property testers for testing if a graph has a vertex cover of size at most k in the sparse model. The first one has two-sided error and is weakly uniform on k. It runs in $O(d/\epsilon)$ time when $k < n/(6d)$, and in $O(1.2738^k + k^2 d)$ time otherwise. The second one has one-sided error and is also weakly uniform on k. It runs in $O(kd/\epsilon)$ time when $k < \epsilon n/4$, and in $O(1.2738^k + k^2/\epsilon)$ time otherwise. Our results reveal the fact that testing if a graph has a small vertex cover can be quite efficient. Next, for testing if a graph has treewidth at most k in the sparse model, we give a $2^{d^{O(kd^3/\epsilon^2)}}$ parameterized property tester and another $d^{(k/\epsilon)^{O(k^2)}} + 2^{\text{poly}(k,d,1/\epsilon)}$ parameterized property tester, both of which have two-sided error. Compared with the $O(2^{\text{poly}(1/\epsilon)})$ property tester in [83] for minor-closed properties, our parameterized property testers are not only uniform on k, but also simpler since they do not need to know the obstruction set (i.e., the set of forbidden minors) of this property. This part of results on parameterized property testing can also refer to the manuscript [42].

We summarize our results in Table 1.3 and 1.4 as follows.

1.5 Dissertation Organization

We give a brief overview of the coming chapters in this section.

In Chapter 2, we introduce the background on evolutionary tree reconstruction and then define the Minimum Quartet Inconsistency problem. For the study of its parameterized complexity, we present three efficient fixed-parameter algorithms of time complexity $O(3.0446^k n + n^4)$, $O(2.0162^k n^3 + n^5)$, and $O^*((1+\varepsilon)^k)$, respectively, where $\varepsilon > 0$ is an arbitrarily small constant.

In Chapter 3, we formulate the problem of determining if a set of quartet topologies is consistent with an evolutionary tree as a combinatorial property, which is called *tree-consistency* of a set of quartet topologies. When the input set consists

Property (Problem)	PC	PT	PPT
MQI	$O(3.0446^k n + n^4)$ **[43]**$\star$ $O(2.0162^k n^3 + n^5)$ **[43]**$\star$ $O^*((1+\varepsilon)^k)$ **[43]**$\star$	–	–
$\mathcal{P}_{tree}$	–	$O(n^3/\epsilon)$ **[44]**#	$O(1.7321^k k n^3/\epsilon)$ **[45]**$\star\star$
$\mathcal{P}_{VC\leq k}$	$O(1.2738^k + kn)$ [48]	$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\} O(\mathrm{poly}(1/\epsilon))$ 2's [11]†	$O(d/\epsilon)$ **[42]**‡♮ $O(kd/\epsilon)$ **[42]**‡♭
$\mathcal{P}_{VC\leq\rho\cdot n}$	–	$\Omega(\sqrt{n})$ [76]‡	
$\mathcal{P}_{tw\leq k}$	$2^{\Theta(k^3)} \cdot k^{O(1)} \cdot n$ [28]	$2^{\mathrm{poly}(1/\epsilon)}$ [83]‡	$2^{d^{O(kd^3/\epsilon^2)}}$ **[42]**‡ $d^{(k/\epsilon)^{O(k^2)}} + 2^{\mathrm{poly}(k,d,1/\epsilon)}$ **[42]**‡

Table 1.3: A summary of our contributions. References in boldface, i.e., **[42–45]**, are our results. PC: parameterized complexity; PT: property testing; PPT: parameterized property testing; $\mathcal{P}_{tree}$: tree-consistency of quartet topologies; $\mathcal{P}_{VC\leq k}$: the property of having a vertex cover of size at most k; $\mathcal{P}_{VC\leq\rho n}$: the property of having a vertex cover of size at most ρn for a constant $\rho \in (0,1)$; $\mathcal{P}_{tw\leq k}$: the property of having treewidth at most k; '$\star$': the parameter k stands for the number of quartet errors; '$\star\star$': the parameter k stands for the number of missing quartets; '#': the input set of quartet topologies is complete; '♮': complexity for $k < n/(6d)$; '♭': complexity for $k < \epsilon n/4$; '†': the result is in the dense model; '‡': the result is in the sparse model.

Property	Sublinear	Testable (easily)	Non-adaptive	1/2-sided error	uniform
$\mathcal{P}_{tree}$	Yes	? (?)	Yes	1	Yes
$\mathcal{P}_{VC\leq k}$	Yes	Yes (no)	No	1	weakly
$\mathcal{P}_{tw\leq k}$	Yes	Yes (?)	No	2	Yes

Table 1.4: A summary of the characteristics of our parameterized property testers. Here "sublinear" is with respect to the input (domain) size.

of exactly one topology for every quartet over an n-taxon set, we prove that there are instances which are ϵ-far from being tree-consistent. Then, we give an $O(n^3/\epsilon)$ property tester for this property. In the end of this chapter, we discuss about the difficulty of dealing with the testing for incomplete set of quartet topologies.

In Chapter 4, we extend the result in Chapter 3. We show that, when we are given an integer $k \geq 0$ which serves as an upper bound on the number of missing quartets with respect to Q, there exists an $O(3^k kn^3/\epsilon)$ parameterized property tester, which is non-adaptive, one-sided-error and uniform on k, for testing if Q is tree-consistent. By carefully enumerating all the possible topologies of the missing quartets which make the set of topologies of all the quartets over S tree-consistent, we obtain another $O(1.7321^k kn^3/\epsilon)$ parameterized property tester, which is also non-adaptive, one-sided-error and uniform on k.

In Chapter 5, we consider parameterized property testing for graph properties. We clarify that there are properties that are trivial to test when the parameter k is small and properties that are parameterized easily testable. Then, we focus on the property of having a vertex cover of size at most k and the property of having treewidth at most k in the sparse model. For testing if a graph has a vertex cover of size at most k in the sparse model, we present a simple adaptive parameterized property tester with two-sided error, which is weakly uniform on k and runs in $O(d/\epsilon)$ time when $k < n/(6d)$, and another one with one-sided error, which is also weakly uniform on k and runs in $O(kd/\epsilon)$ time when $k < \epsilon n/4$. For testing if a graph has treewidth at most k in the sparse model, we present two parameterized property testers of time complexity $2^{d^{O(kd^3/\epsilon^2)}}$ and $d^{(k/\epsilon)^{O(k^2)}} + 2^{\mathrm{poly}(k,d,1/\epsilon)}$, respectively, both of which are uniform on k.

Finally, in Chapter 6 we end the dissertation with concluding remarks and suggestions for future work.

Chapter 2

Fixed-Parameter Algorithms for Minimum Quartet Inconsistency

Determining the evolutionary relationship of a set of taxa is a very essential topic in computational biology. In order to model such relationships, *evolutionary trees* (or *phylogenetic tree*) are widely considered. Roughly speaking, an evolutionary tree represents the course of evolution for a set of taxa over time. In an evolutionary tree, the leaves represent the taxa and the internal nodes represent the ancestors. Building an evolutionary tree for all the taxa has been regarded as a crucial and fundamental problem in computational biology.

As mentioned in [46], we assume that an evolutionary tree is bifurcating (i.e., binary), that is, each internal node of the tree (except the root) is of degree 3. This assumption is due to the reason that events of taxon divergence are usually rare in practical cases, and multifurcations can be viewed as aggregates of bifurcations in some circumstances [114]. Rooted and unrooted evolutionary trees are both studied, however, construction of unrooted evolutionary trees is mostly considered. One crucial reason is that for the same set of leaves, there are fewer unrooted evolutionary trees than rooted ones (See [66] for further discussions). In this dissertation, we focus on unrooted evolutionary trees.

In practical cases, one can only analyze the evolutionary relation of a small set of taxa at one time. Hence, in order to observe the whole evolution course of all the taxa, one has to construct an evolutionary tree from a set of small subtrees. However, in reality some of the given small trees may be erroneous, so an evolutionary tree consistent with all of them may not exist. Moreover, determining if such an evolutionary tree exists is proved to be **NP**-complete [110]. Therefore, one turns to seek for an evolutionary tree consistent with as many of the input small trees as possible. For practical and theoretical advantages, *quartet methods*, where the input consists of a set of unrooted four-leaf trees (i.e., *quartet topologies*), are extensively used for this kind of optimization problem in the past three decades [17, 20, 23, 24, 41, 43–46, 50, 62, 78, 85, 86, 109, 110, 112]. This method is based on the fact that any evolutionary tree can be uniquely characterized by its set of induced quartet topologies [37, 50].

We focus on the *Minimum Quartet Inconsistency* (MQI) problem and its parameterized complexity. Roughly speaking, the MQI problem asks for an evolutionary tree from a set of small four-leaf unrooted trees, such that the number of inconsistent small trees is minimized. Assume that exactly one quartet topology for each set of four taxa is given. Provided with a parameter k denoting the upper bound on the number of inconsistent small four-leaf unrooted trees, we show that the MQI problem admits efficient fixed-parameter algorithms.

In Sect. 2.1, we introduce the MQI problem as well as basic terminologies and the related work. In Sect. 2.2, we introduce the strategy of designing fixed-parameter algorithms using the *depth-bounded search tree* . Then, we provide three efficient fixed-parameter algorithms for the MQI problem. The first one, presented in Sect. 2.3, is an $O(3.0446^k n + n^4)$ fixed-parameter algorithm, which is designed using the depth-bounded search tree. The second one, which is obtained by extending the first one, is an $O(2.0162^k n^3 + n^5)$ fixed-parameter algorithm and presented in Sect. 2.4. In Sect. 2.5, we present an $O^*((1+\varepsilon)^k)$ fixed-parameter algorithm, where $\varepsilon > 0$ is an arbitrarily small constant. The running time of the third algorithm has an exponential term with an arbitrarily small base, which can be very close to 1, yet its polynomial factor grows quickly as the base of the exponential term decreases.

2.1 The Minimum Quartet Inconsistency Problem

2.1.1 Preliminaries and terminologies

Let S be a set of n taxa. An evolutionary tree T over S is an *unrooted*, *leaf-labeled* tree such that the leaves of T are bijectively labeled by the taxa in S, and each internal node of T has degree three. A *quartet* is a set of four taxa in S. The *quartet* *topology* for $\{a, b, c, d\}$ induced by T is the path structure connecting a, b, c, and d in T (see Fig. 2.1 for an illustration). Equivalently, we say that $\{a, b, c, d\}$ has the quartet topology $[ab|cd]$ with respect to T if and only if the path on T from a to b does not share any vertex with that from c to d. In this dissertation, a quartet $\{a, b, c, d\}$ is restricted to have three possible topologies $[ab|cd]$, $[ac|bd]$, and $[ad|bc]$ (see Fig. 2.2), which are the possible bipartitions of $\{a, b, c, d\}$ (hence $[ab|cd]$, $[ba|cd]$, $[ab|dc]$, $[ba|dc]$, $[cd|ab]$, $[dc|ab]$, $[cd|ba]$, $[dc|ba]$ are regarded the same).

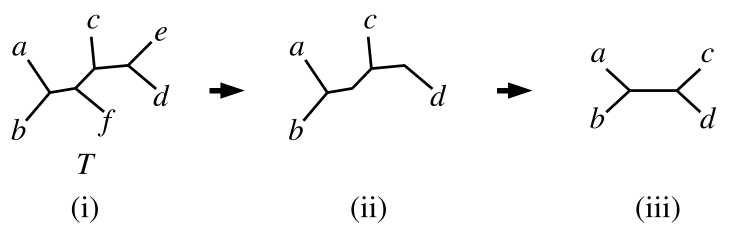

Figure 2.1: (i) An evolutionary tree T; (ii) The path structure connecting a, b, c, d in T; (iii) The quartet topology of $\{a, b, c, d\}$ induced by T.

We denote by Q_T the set of quartet topologies induced by an evolutionary tree T. A set of quartet topologies Q is said to be *complete* (with respect to S) if Q contains exactly one topology for every quartet in S. We say that Q is *tree-consistent* [17] if there exists an evolutionary tree T such that $Q \subseteq Q_T$. Furthermore, we say that Q is *tree-like* [17] if $Q = Q_T$ for some evolutionary tree T. For example, if $S = \{a, b, c, d, e, f\}$ and $Q = \{[ab|cd], [ab|ce], [ab|cf], [ab|de], [ab|df], [ab|ef], [ac|de], [af|cd], [af|ce], [af|de], [bc|de], [bf|cd], [bf|ce], [bf|de], [cf|de]\}$, then Q is tree-like since it is exactly the set of quartet topologies induced by T in Fig. 2.1 (i). Let Υ be the set of all tree-like sets of quartet topologies over S. We call $\min_{Q^* \in \Upsilon} |Q \setminus Q^*|$ the *error number* of Q. We call the quartet topologies in $Q \setminus Q^*$ the *quartet errors* of Q if $|Q \setminus Q^*|$ equals to the error number of Q for $Q^* \in \Upsilon$. Note that the number $|Q \setminus Q^*|$ is equal to $|Q^* \setminus Q|$ since Q and Q^* are complete (if a quartet has a topology is in $Q \setminus Q^*$ then there must be a different one of this quartet in $Q^* \setminus Q$).

In the following we formally state the MQI problem. By introducing a parameter k to the MQI problem, we obtain its parameterized version, which is called *parameterized MQI* problem for short.

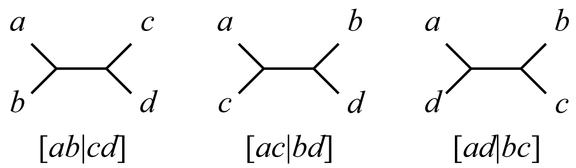

Figure 2.2: Three topologies for a quartet $\{a, b, c, d\}$.

The Minimum Quartet Inconsistency Problem (MQI):
Input: A complete set of quartet topologies Q over an n-taxon set S.
Task: Construct an evolutionary tree T on S such that the number of quartet errors of Q with respect to Q_T is minimized.

The Parameterized Minimum Quartet Inconsistency Problem (parameteried MQI):
Input: A complete set of quartet topologies Q over an n-taxon set S, and an integer k.
Task: Determine if there exists an evolutionary tree T on S such that the number of quartet errors of Q with respect to Q_T is at most k.

2.1.2 Related work

The *Quartet Compatibility Problem* (QCP) is to determine if there exists an evolutionary tree T on S satisfying all quartet topologies Q. The QCP problem can be solved in polynomial time whenever Q is complete [62], but it becomes **NP**-complete when Q is not necessarily complete [110]. From now on we consider the case that Q is complete. The optimization problem, called the *Maximum Quartet Consistency* problem (MQC), is a dual problem to the MQI problem. The MQC problem is to construct an evolutionary tree T on S to satisfy as many quartet topologies of Q as possible. The MQC problem and the MQI problem are both **NP**-hard [24], however, the MQC problem admits a *polynomial time approximation scheme* (PTAS) [86], while the best approximation ratio found so far for the MQI problem is $O(n^2)$ [85]. Ben-Dor et al. gave an $O(3^n n^4)$ algorithm to solve the MQI problem by dynamic programming [20]. For the case that Q has less than $(n-3)/2$ quartet errors, Berry et al. [24] devised an $O(n^4)$ algorithm for the MQI problem. Furthermore, if Q has at most cn quartet errors, Wu et al. [117] compute the optimal solution for the MQI problem in $O(n^5 + 2^{4c} n^{12c+2})$ time, where c is some positive constant. While this is a polynomial time algorithm, the degree of the polynomial in the run-time grows quickly. Therefore parameterized algorithms are faster for practical values of k and n.

As to the parameterized complexity of the MQI problem, Gramm and Niedermeier proved that the parameterized MQI problem is *fixed parameter tractable* [78], and they proposed a $O(4^k n + n^4)$ fixed parameter algorithm [78]. In [116], Wu et al. presented a lookahead branch-and-bound algorithm for the MQC problem which runs in time $O(4^{k'} n^2 k' + n^4)$, where k' is an upper bound on the number of quartet errors of Q.

2.2 The Main Approach: Depth-Bounded Search Tree

In this chapter, we utilize the strategy of *depth-bounded search trees*, which is one of the most important concepts in design and analysis of fixed-parameter algorithms [60, 98]. A depth-bounded search tree algorithm works recursively. The number of recursions is the size of the corresponding search tree. Such an algorithm explores an optimal solution for an **NP**-hard problem by performing systematic exhaustive search in a search tree. The depth of the search tree is bounded by a parameter. Concerning the complexity analysis of the algorithm, we have to determine an upper bound on the size of the corresponding search depending on the structure of algorithm recursions. Note that we concentrate on linear recurrences with constant coefficients here. The basic definitions and results which are used in this dissertation are listed as follows.

Definition 2.1 ([98])**.** Given a problem $\mathbb{P}$ with parameter k. If an algorithm solves $\mathbb{P}$ and calls itself recursively for subproblems with parameters $k-d_1, k-d_2, \ldots, k-d_i$, then $(d_1, d_2, \ldots, d_i)$ is called the *branching vector* of recursion of the algorithm.

Actually, the branching vector $(d_1, d_2, \ldots, d_i)$ corresponds to the recurrence $T_k = T_{k-d_1} + T_{k-d_2} + \ldots + T_{k-d_i}$. In addition, we assume that $T_0 = T_1 = \ldots = T_{d'-1} = 1$, where $d' = \min\{d_1, \ldots, d_i\}$, for the boundary condition of the recurrence. Note that T_k corresponds to the number of leaves in the search tree, and the number of nodes in the search tree is at most $2T_k$.

Definition 2.2 ([98]). Given a branching vector $\mathbf{v} = (d_1, d_2 \ldots, d_i)$ of some recursion, the *characteristic polynomial* of $\mathbf{v}$ is

$$z^d - z^{d-d_1} - z^{d-d_2} - \ldots - z^{d-d_i},$$

where d is defined to be $\max\{d_1, d_2, \ldots, d_i\}$. Furthermore, we call α the *characteristic root* of the characteristic polynomial if $\alpha^d = \alpha^{d-d_1} + \alpha^{d-d_2} + \ldots + \alpha^{d-d_i}$.

Definition 2.3 ([80, 100]). Given a branching vector $\mathbf{v} = (d_1, d_2, \ldots, d_i)$ of some recursion, the *reflected characteristic polynomial* of $\mathbf{v}$ is $1 - z^{d_1} - z^{d_2} - \ldots - z^{d_i}$.

For example, assume that we have a recurrence $T(k) = 2T(k-1) + T(k-3) + T(k-5)$. The branching vector of the recurrence is $(1, 1, 3, 5)$ and its characteristic polynomial and reflected characteristic polynomial are $z^5 - 2z^4 - z^2 - 1$ and $1 - 2z - z^3 - z^5$ respectively. The characteristic root of the characteristic polynomial is $2.2392\ldots$.

Let α be the characteristic root of the characteristic polynomial $z^d - z^{d-d_1} - z^{d-d_2} - \ldots - z^{d-d_i}$. It is well known that the root of the reflected characteristic polynomial $1 - z^{d_1} - z^{d_2} - \ldots - z^{d_i}$ is $1/\alpha$ [80, 100].

Theorem 2.1 ([80, 98, 100]). *A depth-bounded search tree with branching vector $(d_1, d_2, \ldots, d_i)$ and its root labeled with parameter k has size $k^{O(1)} \cdot \alpha^k$, where α is the greatest characteristic root the corresponding characteristic polynomial. Furthermore, if α is not a multiple root, then the size of the search tree is $\Theta(\alpha^k)$.*

Remarks. Let $\mathbf{v} = (d_1, d_2, \ldots, d_i)$, where $d_1, d_2, \ldots, d_i > 0$ and $i > 1$, be a branching vector. Let $f(z) = z^d - z^{d-d_1} - z^{d-d_2} - \ldots - z^{d-d_i}$, where $d = \max\{d_1, d_2, \ldots, d_i\}$, be the characteristic polynomial of $\mathbf{v}$. For the analysis of the corresponding search tree size, we only care about the roots of $f(z)$ which are greater than 1, hence we focus on the polynomial $g(z) = f(z)/z^d = 1 - z^{-d_1} - z^{-d_2} - \ldots - z^{-d_i}$. Note that the derivative of $g(z)$ is $g'(z) = d_1 z^{-d_1-1} + d_2 z^{-d_2-1} + \ldots + d_i z^{-d_i-1}$. Since $d_1, d_2, \ldots, d_i > 0$, we have $g'(z) > 0$ for all $z > 0$ so that $g(z)$ is monotonically increasing in $(0, \infty)$. Since it is clear that $g(1) < 0$ and $g(z)$ is monotonically increasing in $(1, \infty)$, there must be exactly one root $\alpha > 1$ of $g(z)$, which is simple (i.e., α is not a multiple root) due to the fact that $g'(\alpha) > 0$. Thus, there is also exactly one root

$1/\alpha$ of the corresponding reflected characteristic polynomial $1 - z^{d_1} - z^{d_2} - \ldots - z^{d_i}$, where $0 < 1/\alpha < 1$. In the remainder of this dissertation, each branching vector has positive entries. Thus, whenever a root $\alpha > 1$ (resp., $0 < 1/\alpha < 1$) of a characteristic polynomial (resp., a reflected characteristic polynomial) is found, Theorem 2.1 imples that the size of the corresponding search tree is $\Theta(\alpha^k)$.

For simplicity, we call the base of the exponentially growing function in Theorem 2.1, i.e., α, the *branching number*. Let $\rho(\mathbf{v})$ denote the branching number corresponding to a branching vector $\mathbf{v}$. Note that the ordering of a branching vector does not affect the corresponding branching number. The following theorem concerns about the relation between a branching vector its corresponding branching number.

Theorem 2.2. *Let* $\mathbf{v} = (d_1, d_2, \ldots, d_i)$ *and* $\mathbf{v}' = (d'_1, d'_2, \ldots, d'_i)$ *be two branching vectors, where* $d_j \le d'_j$ *for* $1 \le j \le i$*, then* $\rho(\mathbf{v}) \ge \rho(\mathbf{v}')$*.*

Proof. The reflected characteristic polynomial of $\mathbf{v}$ and $\mathbf{v}'$ are $1 - \sum_{j=1}^{i} z^{d_j}$ and $1 - \sum_{j=1}^{i} z^{d'_j}$ respectively. Let z_0 and z'_0 be the roots of $1 - \sum_{j=1}^{i} z^{d_j}$ and $1 - \sum_{j=1}^{i} z^{d'_j}$ respectively, then we have $\sum_{j=1}^{i} {z_0}^{d_j} = 1$ and $\sum_{j=1}^{i} {z'_0}^{d_j} = 1$. Since $z_0 < 1$ and $d_j \le d'_j$ for $1 \le j \le i$, we have ${z_0}^{d'_j} \le {z_0}^{d_j}$ for all $1 \le j \le i$, and hence $\sum_{j=1}^{i} {z_0}^{d'_j} \le 1$. Thus z'_0 must be greater than or equal to z_0. Therefore, $\rho(\mathbf{v}) = 1/z_0 \ge 1/z'_0 = \rho(\mathbf{v}')$. □

Based on the above definitions and observations, we devise a C language program that can calculate the branching number of an input branching vector with positive entries. Refer to [89] for the program as well as its source code.

2.3 An $O(3.0446^k n + n^4)$ Fixed-Parameter Algorithm

2.3.1 Quintets and tree-consistency

A *quintet* is a set of five taxa in S. Let Q denote a complete set of quartet topologies over S. Clearly, Q is of size $\binom{n}{4}$. We say that a quintet $\{a, b, c, d, e\} \subseteq S$ is *resolved* with respect to Q if the set of quartet topologies over $\{a, b, c, d, e\}$ in Q is tree-like. Otherwise, we say that $\{a, b, c, d, e\}$ is *unresolved* with respect to Q. Similar to the quartet topology, the *quintet topology* of a quintet $\{a, b, c, d, e\}$ induced by an evolutionary tree T is the path structure connecting a, b, c, d, and e in T.

Without loss of generality, assume that we have $[ab|cd]$ induced by T, then there are five quintet topologies for the quintet $\{a, b, c, d, e\}$ induced by T since there are five positions for inserting e into the tree structure of $[ab|cd]$. Since there are three different topologies for the quartet $\{a, b, c, d\}$, there are fifteen quintet topologies for a quintet $\{a, b, c, d, e\}$ (see Fig. 2.3).

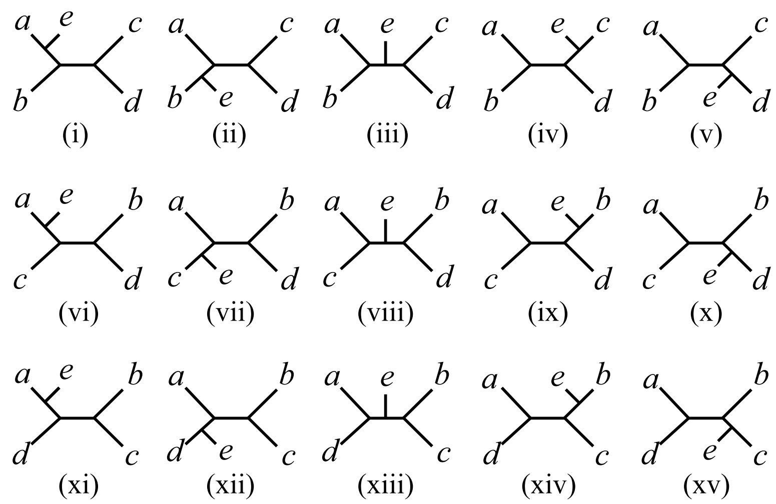

Figure 2.3: The fifteen topologies for a quintet $\{a, b, c, d, e\}$.

We say that a set of quartet topologies Q' over S *involves a taxon* f if there exists at least one quartet topology $t = [v_1 v_2 | v_3 v_4] \in Q'$, where $v_1, v_2, v_3, v_4 \in S$, such that $f = v_i$ for some $i \in \{1, 2, 3, 4\}$. If a set of quartet topologies is not tree-consistent, we say that it has a *conflict* [78]. We say that a set of three topologies has a *local conflict* [78] if it is not tree-consistent. Concerning the connection between local conflicts and tree-likeness, Gramm and Niedermeier proved the following lemma and theorem.

Lemma 2.1 ([78]). *A set of three quartet topologies, each of which comes from different quartets, is tree-consistent if it involves more than five taxa.*

Theorem 2.3 ([78]). *Given a set of taxa S and a complete set of quartet topologies Q over S, and some taxon $f \in S$, then Q is tree-like if and only if every set of three quartet topologies in Q that involves f has no local conflict.*

The following lemma relates local conflicts with unresolved quintets.

Lemma 2.2. *Assume that* $\mathbf{q} \subseteq S$ *is a quintet such that* $f \in \mathbf{q}$ *and let* $Q_{\mathbf{q}} \subseteq Q$ *denote the set of quartet topologies of quartets in* $\mathbf{q}$. *Then* $\mathbf{q}$ *is resolved if and only if every set of three quartet topologies in* $Q_{\mathbf{q}}$ *has no local conflict.*

***Proof*.** Recall that $\mathbf{q}$ is resolved if and only if there exists an evolutionary tree T with leaf set $\mathbf{q}$ such that $Q_{\mathbf{q}} = Q_T$, i.e., $Q_{\mathbf{q}}$ is tree-consistent. Furthermore, we can derive by Theorem 2.3 that $Q_{\mathbf{q}}$ is tree-consistent (or tree-like when regarding $\mathbf{q}$ as the taxon set) if and only if every set of three quartet topologies in $Q_{\mathbf{q}}$ has no local conflict. Therefore the lemma follows. □

By Lemma 2.1 and Lemma 2.2, we observe the relation between tree-likeness and resolved quintets and Theorem 2.4 follows. Actually, this theorem can be easily derived from Bandelt and Dress' result [17].

Theorem 2.4 ([17])**.** *Given a set of taxa* S*, a complete set of quartet topologies* Q *over* S*, and some taxon* $f \in S$*, then* Q *is tree-like if and only if every quintet containing* f *is resolved.*

There are $\binom{\binom{5}{4}}{3} = 10$ sets of three quartets with respect to a quintet $\{a, b, c, d, e\}$. Checking whether a set of three quartet topologies has a local conflict requires only constant time [78]. It is then clear that checking whether a quintet is resolved requires only constant time. With a taxon $f \in S$ which is fixed, there are $\binom{n-1}{4}$ quintets containing f. Thus we have the following theorem.

Theorem 2.5. *Given a set* S *of taxa, some taxon* $f \in S$*, and a complete set* Q *of quartet topologies, then all unresolved quintets involving* f *can be found in* $O(n^4)$ *time.*

Let $\prec$ be a total order on the taxon set S. Without loss of generality, every set of l taxa is represented according to $\prec$. That is, we denote a set of taxa by $\{s_1, s_2, \ldots, s_l\}$ if $s_1 \prec s_2 \ldots \prec s_l$. A quartet topology is represented by $[s_1 s_2 | s_3 s_4]$ if $s_1 \prec s_3$, $s_1 \prec s_2$, and $s_3 \prec s_4$. For the three possible topologies of a quartet, we denote them by *type* 0, 1, and 2 according to $\prec$. Consider a quartet $\{a, b, c, d\} \subset S$

as an example. If $a \prec b \prec c \prec d$, we denote $[ab|cd]$ by 0, $[ac|bd]$ by 1, and $[ad|bc]$ by 2.

Let $\prec_l$ be the lexicographic order on the Cartesian product of l's S according to the total order $\prec$. For a quintet $\{s_1, s_2, s_3, s_4, s_5\}$, where $s_1 \prec s_2 \prec s_3 \prec s_4 \prec s_5$, we define its *topology vector* to be an ordered sequence $(r_1, r_2, r_3, r_4, r_5)$, where r_1, r_2, r_3, r_4, and r_5 are the types of quartet topologies of $\{s_1, s_2, s_3, s_4\}$, $\{s_1, s_2, s_3, s_5\}$, $\{s_1, s_2, s_4, s_5\}$, $\{s_1, s_3, s_4, s_5\}$, and $\{s_2, s_3, s_4, s_5\}$ respectively (i.e., the quartets in the order of $\prec_5$). For example, consider a quintet $\{a, b, c, d, e\} \subseteq S$, where $a \prec b \prec c \prec d \prec e$. Assume that $[ab|cd]$, $[ae|bc]$, $[ab|de]$, $[ae|cd]$, and $[bd|ce]$ are in Q, then the topology vector of $\{a, b, c, d, e\}$ is $(0, 2, 0, 2, 1)$. Recall that there are 15 possible quintet topologies for a quintet $\{s_1, s_2, s_3, s_4, s_5\}$. We denote by $\mathcal{V}$ the set of topology vectors of all the possible quintet topologies of a quintet, then we have

$$\mathcal{V} = \left\{ \begin{array}{lllll} (0,0,0,0,0), & (1,1,0,0,0), & (2,2,0,0,0), & (2,2,1,1,0), & (2,2,2,2,0), \\ (0,0,0,1,1), & (2,0,1,1,1), & (1,0,2,1,1), & (1,1,2,0,1), & (1,2,2,2,1), \\ (0,0,0,2,2), & (0,2,2,2,2), & (0,1,1,2,2), & (1,1,1,0,2), & (2,1,1,1,2). \end{array} \right\}.$$

Note that the size of $\mathcal{V}$ is far less than the number of possible topology vectors of a quintet, which is $3^5 = 243$.

2.3.2 The algorithm

Our first fixed-parameter algorithm is called FPA1-MQI, which runs recursively. The concepts of the algorithm are as follows. We build a list of unresolved quintets $\mathcal{C}_f$ containing some fixed taxon f and the list $\mathcal{V}$ of topologies vectors of possible quintet topologies for a quintet as preprocessing steps. In each recursion, the algorithm selects an unresolved quintet $\mathbf{q} = \{a, b, c, d, e\} \in \mathcal{C}_f$ arbitrarily and then tries to make $\mathbf{q}$ resolved by the procedure update according to all the possible fifteen quintet topologies of $\mathbf{q}$.

FPA1-MQI$(Q, k, \mathcal{C}_f)$
/* Q: a complete set of quartet topologies; k: an integer parameter;
$\mathcal{C}_f$: a list of unresolved quintets. */
begin
if $\mathcal{C}_f$ is empty and $k \geq 0$ **then**
  return ACCEPT;
else if $k \leq 0$ **then**
  return
end if
extract an unresolved quintet $\mathbf{q}$ from $\mathcal{C}_f$;
for each $\mu \in \mathcal{V}$ **do**
  $(Q', \mathcal{C}'_f, k') \leftarrow$ update$(Q, \mathcal{C}_f, \mathbf{q}, \mu, k)$;
  FPA1-MQI $(Q', k', \mathcal{C}'_f)$;
end for
end

Algorithm 2.1: FPA1-MQI: an $O(3.0446^k n + n^4)$ algorithm for the parameterized MQI problem.

Recall that each topology vector $\mu \in \mathcal{V}$ represents a quintet topology of a quintet. The procedure update changes quartet topologies according to the quartet topologies which μ stands for, and updates the set $\mathcal{C}_f$ and the parameter k to be $\mathcal{C}'_f$ and k' respectively. For example, assume that we have $[ab|cd]$, $[ae|bc]$, $[ab|de]$, $[ae|cd]$, and $[bd|ce]$ in Q for the quintet $\{a, b, c, d, e\}$ (the corresponding topology vector is then $(0, 2, 0, 2, 1)$), and assume that $\mu = (2, 1, 1, 1, 2)$. The procedure update changes these quartet topologies to $[ad|bc]$, $[ac|be]$, $[ad|be]$, $[ad|ce]$, and $[be|cd]$ respectively, according to μ, and these quartets are marked so that their topologies will not be changed again. However, if there is a branch node in the search tree such that some quartet topology, whose corresponding quartet has been marked, must be changed in all the possible 15 branches to make an unresolved quintet resolved, the algorithm stops branching here and just returns (since all the possible changes of topologies of this quartet have been already considered by the algorithm to make some certain quintet resolved when this quartet was marked). Let Q_μ denote the set of quartet topologies changed according to μ. The procedure update obtains the updated inconsistent quintet set $\mathcal{C}'_f$ by removing the newly resolved quintets and adding the newly unresolved quintets from $\mathcal{C}_f$, and gets the updated parameter k' by letting $k' = k - |Q_\mu|$.

By Theorem 2.4, we know $\mathcal{C}_f$ is empty if and only if the set of quartet topologies is tree-like. Algorithm FPA1-MQI branches in all possible ways to eliminate each unresolved quintet in $\mathcal{C}_f$ and it changes at most k quartet topologies from the root to each branch node in the search tree. furthermore, the algorithm returns ACCEPT if and only if $\mathcal{C}_f = \emptyset$ and at most k quartet topologies are changed, thus it is correct.

2.3.3 Time complexity

Building lists $\mathcal{C}_f$ and $\mathcal{V}$.

Building $\mathcal{C}_f$ requires $O(n^4)$ time by Theorem 2.5. Furthermore, building $\mathcal{V}$ requires only constant time.

The recursive structure of Algorithm FPA1-MQI.

The algorithm works as a depth-bounded search tree. Each tree node has 15 branches and each branch corresponds to a quintet topology. The root of the search tree is labeled by k. Let us denote the size of the search tree rooted at a node labeled r to be the $T(r)$. For each $\mu \in \mathcal{V}$, we have $T(r) = \sum_{\mu \in \mathcal{V}} T(r - |Q_\mu|)$, i.e., the branching vector is $(|Q_\mu|)_{\mu \in \mathcal{V}}$. Since there are 243 possible topology vectors of a quintet but 15 of them are in $\mathcal{V}$, we have 228 possible branching vectors as well as 228 branching numbers. Table 2.1 lists the branching vectors and the corresponding branching numbers (refer to Appendix C for all the 243 branching vectors as well as their branching numbers).

Consider the first row in Table 2.1 for an illustration. In this case, the algorithm selects a quintet $\mathbf{q} = \{a, b, c, d, e\}$ which has induced quartet topologies $[ab|cd]$, $[ac|be]$, $[ae|bd]$, $[ad|ce]$, and $[bc|de]$ in Q. By comparing its corresponding topology vector $(0, 1, 2, 1, 0)$ with each topology vector $\mu \in \mathcal{V}$, we obtain that the numbers of quartet topologies changed by Algorithm FPA1-MQI are 3, 3, 4, 3, 3, 3, 4, 3, 3, 4, 4, 3, 3, 4, and 3 respectively. Hence we have a branching vector $(3, 3, 4, 3, 3, 3, 4, 3, 3, 4, 4, 3, 3, 4, 3)$ and then we can compute a branching number between 2.3004 and 2.3005. It can be derived that the branching number in the worst case is greater than 3.0445 and less than 3.0446. Thus the size of $T(k)$ is $O(3.0446^k)$.

Table 2.1: Some possible branching vectors and branching numbers of FPA1-MQI.

topology vector	branching vector	branching number
$(0,1,2,1,0)$	$(3,3,4,3,3,3,4,3,3,4,4,3,3,4,3)$	2.30042...
$(0,0,1,0,1)$	$(2,4,4,4,5,2,2,3,3,4,3,4,3,3,4)$	2.46596...
$(0,0,1,0,2)$	$(2,4,4,4,5,3,3,4,4,5,2,3,2,2,3)$	2.54314...
$(0,0,1,0,0)$	$(1,3,3,3,4,3,3,4,4,5,3,4,3,3,4)$	2.55234...
$(0,0,1,1,2)$	$(3,5,5,3,5,2,2,3,5,5,2,3,2,3,2)$	2.67102...
$(0,0,0,0,1)$	$(1,3,3,5,5,1,3,3,3,4,2,4,4,4,5)$	3.04454...

The procedure update.

For $\mu \in \mathcal{V}$, since there are $n-4$ quintets involving a fixed quartet, there are at most $|Q_\mu|(n-4)$ quintets involving quartet topologies in Q_μ. Thus the procedure update runs only in $O(n)$ time.

From the above analysis, we derive that the time complexity of Algorithm FPA1-MQI is $O(3.0446^k n + n^4)$. Thus the following theorem follows.

Theorem 2.6. *There exists an $O(3.0446^k n + n^4)$ fixed-parameter algorithm for the parameterized minimum quartet inconsistency problem.*

2.4 An $O(2.0162^k n^3 + n^5)$ Fixed-Parameter Algorithm

2.4.1 Sextets with siblings

Two taxa a, b are *siblings* on an evolutionary tree T if a and b are both adjacent to the same internal vertex in T. Here we consider the sextet topologies of the sextet $\{a, b, w, x, y, z\}$ where a, b are siblings. It is clear that there are fifteen possible sextet topologies with siblings a, b for a sextet $\{a, b, w, x, y, z\}$ (see Fig. 2.4 for an illustration)

Assume that s_1, s_2 are siblings in an evolutionary tree over S, and hence that we have 15 sextet topologies for the sextet $\{s_1, s_2, s_3, s_4, s_5, s_6\} \subseteq S$. There are $\binom{6}{4} = 15$ quartets with respect to the sextet $\{s_1, s_2, s_3, s_4, s_5, s_6\}$, yet $\binom{4}{2} = 6$ of them have fixed quartet topologies since s_1, s_2 are siblings. For example, the quartet topology of $\{s_1, s_2, s_3, s_4\}$ must be $[s_1 s_2 | s_3 s_4]$. Given two siblings s_1, s_2, the $\{s_1, s_2\}$*-reduced topology vector* of sextet $\{s_1, s_2, s_3, s_4, s_5, s_6\}$ is an ordered sequence of types of the quartet topologies which are not fixed. For example, consider a sextet

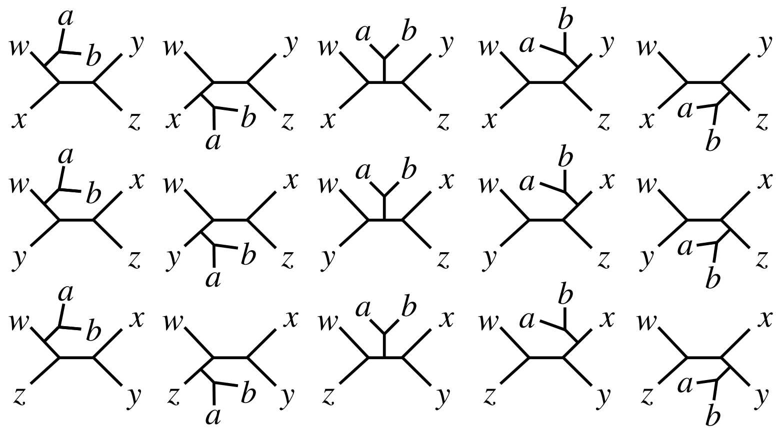

Figure 2.4: The fifteen possible sextet topologies for the sextet $\{a, b, w, x, y, z\}$ with siblings a, b.

$\{a, b, w, x, y, z\} \subseteq S$ with siblings a, b such that $[aw|xy]$, $[ax|wz]$, $[az|wy]$, $[ay|xz]$, $[bw|xy]$, $[bx|wz]$, $[bz|wy]$, $[by|xz]$, and $[wx|yz]$ are in Q. The $\{a, b\}$-reduced topology vector of $\{a, b, w, x, y, z\}$ is $(0, 1, 2, 1, 0, 1, 2, 1, 0)$. Let us denote by $\mathcal{V}_2$ the set of $\{a, b\}$-reduced topology vectors of all possible sextet topologies of $\{a, b, w, x, y, z\}$. Then we have

$$\mathcal{V}_2 = \left\{ \begin{array}{lll} (0,0,0,0,0,0,0,0,0), & (1,1,0,0,1,1,0,0,0), & (2,2,0,0,2,2,0,0,0), \\ (2,2,1,1,2,2,1,1,0), & (2,2,2,2,2,2,2,2,0), & (0,0,0,1,0,0,0,1,1), \\ (2,0,1,1,2,0,1,1,1), & (1,0,2,1,1,0,2,1,1), & (1,1,2,0,1,1,2,0,1), \\ (1,2,2,2,1,2,2,2,1), & (0,0,0,2,0,0,0,2,2), & (0,2,2,2,0,2,2,2,2), \\ (0,1,1,2,0,1,1,2,2), & (1,1,1,0,1,1,1,0,2), & (2,1,1,1,2,1,1,1,2). \end{array} \right\}.$$

2.4.2 The two-siblings-determined minimum quartet inconsistency problem

We define the *two-siblings-determined minimum quartet inconsistency problem* as follows. Given a complete quartet topology set Q over a taxon set S, a parameter k and two taxa $a, b \in S$ as the input, determine whether there exists an evolutionary tree T on which a and b are siblings such that Q_T differs from Q in at most k quartet topologies. We abbreviate this problem as *2SDMQI* for the readers' convenience.

We present a fixed-parameter algorithm called FPA-2SDMQI for the 2SDMQI problem as follows. First, for every $u, v \in S \setminus \{a, b\}$ such that $[ab|uv] \notin Q$, we change the quartet topology of $\{a, b, u, v\}$ to be $[ab|uv]$ and decrease k by 1. Note that $k \leq 0$ at Line 3 of Algorithm FPA-2SDMQI means that the algorithm has to change more than k quartet topologies to make a, b be siblings on an evolutionary tree, so it just returns. Second, we build two lists $\mathcal{C}_a$ and $\mathcal{V}_2$, where $\mathcal{C}_a$ is a list of unresolved quintets containing a while $\mathcal{V}_2$ is a list of $\{a, b\}$-reduced topologies vectors of possible sextet topologies on which a, b are siblings. Then the algorithm calls Algorithm Resolve as a subroutine to resolve all $\{a, b\}$-unresolved sextets by changing at most k quartet topologies.

FPA-2SDMQI($Q, k, \mathcal{C}_a, a, b$)
/* Q: a complete set of quartet topologies; k: an integer parameter;
$\mathcal{C}_f$: a list of unresolved quintets; a, b: two taxa. */
begin
if $\mathcal{C}_a$ is empty and $k \geq 0$ **then**
  return ACCEPT;
else if $k \leq 0$ **then**
  return
end if
for every two taxa $u, v \in S \setminus \{a, b\}$ **do**
  if $k \leq 0$ **then**
    return
  else
    change the quartet topology of $\{a, b, u, v\}$ to be $[ab|uv]$ if $[ab|uv] \notin Q$;
    update $\mathcal{C}_a$ and $k \leftarrow k - 1$;
  end if
end for
Resolve($Q, k, \mathcal{C}_a, a, b$);
end

Algorithm 2.2: FPA-2SDMQI: a fixed-parameter algorithm for the 2SDMQI problem.

Algorithm Resolve works recursively. In each recursion, it arbitrarily selects an unresolved quintet $\mathbf{q}$. It is clear that $\mathbf{q} \cup \{b\}$ is $\{a, b\}$-unresolved. Then Algorithm Resolve tries to make $\mathbf{q} \cup \{b\}$ be $\{a, b\}$-resolved by the procedure update$_2$ according to all the possible 15 sextet topologies of $\mathbf{q} \cup \{b\}$ having a, b as siblings. Similar to the procedure update in Sect. 2.3, we mark the quartets whose topologies are changed, and if there is a branch node in the search tree such that some quartet, which has been marked, must be changed in all the possible 15 branches to make $\mathbf{q} \cup \{b\}$ be $\{a, b\}$-resolved, the algorithm stops branching here and just returns (by the same reason mentioned in Sect. 2.3.2).

Each $\{a, b\}$-reduced topology vector $\nu \in \mathcal{V}_2$ represents a sextet topology of a sextet with siblings a, b. The procedure update_2 changes quartet topologies according to the quartet topologies that ν stands for, marks these quartets so that their topologies will not be changed again, and updates the set $\mathcal{C}_a$ and the parameter k to be $\mathcal{C}'_a$ and k' respectively. We denote by Q_ν the set of quartet topologies changed according to ν. The procedure update_2 gets the updated $\mathcal{C}'_a$ by removing the newly resolved quintets and adding the newly unresolved quintets from $\mathcal{C}_a$, and gets the updated parameter k' by letting $k' = k - |Q_\nu|$. Similar to the analysis of Algorithm FPA1-MQI, we can derive easily that Algorithm FPA-2SDMQI is correct.

Resolve$(Q, k, \mathcal{C}_a, a, b)$
/* Q: a complete set of quartet topologies; k: an integer parameter;
$\mathcal{C}_a$: a list of unresolved quintets; a, b: two taxa. */
begin
1: **if** $\mathcal{C}_a$ is empty and $k \geq 0$ **then**
2:    **return** ACCEPT;
3: **else if** $k \leq 0$ **then**
4:    **return**
5: **end if**
6: extract an unresolved quintet $\mathbf{q}$ from $\mathcal{C}_a$;
7: **if** $b \in \mathbf{q}$ **then**
8:    $\mathbf{q} \leftarrow \mathbf{q} \cup \{s\}$, for some arbitrary taxon $s \notin \mathbf{q}$;
9: **else**
10:    $\mathbf{q} \leftarrow \mathbf{q} \cup \{b\}$;
11: **end if**
12: **for** each $\nu \in \mathcal{V}_2$ **do**
13:    $(Q', \mathcal{C}'_a, k') \leftarrow \mathsf{update}_2(Q, \mathcal{C}_a, \mathbf{q}, \nu, k)$;
14:    Resolve$(Q', k', \mathcal{C}'_a, a, b)$;
15: **end for**
end

Algorithm 2.3: Resolve: a subroutine of FPA-2SDMQI.

Time complexity of nonrecursive steps. Execution of Lines 6–13 in Algorithm FPA-2SDMQI takes $O(n^2)$ time. Building $\mathcal{C}_a$ requires $O(n^4)$ time by Theorem 2.5. Furthermore, it is obvious that building $\mathcal{V}_2$ costs only constant time.

Time complexity of the recursive structure of Algorithm FPA-2SDMQI. The algorithm (i.e., Algorithm Resolve) again works as a depth-bounded search tree. Each tree node has 15 branches and each branch corresponds to a sextet topology with siblings a, b. The root of the search tree is labeled by k. Let us denote the size of the search tree rooted at a node labeled r to be the $T_2(r)$. For each $\nu \in \mathcal{V}_2$, we have $T_2(r) = \sum_{\nu \in \mathcal{V}_2} T_2(r - |Q_\nu|)$, that is, the branching vector is $(|Q_\nu|)_{\nu \in \mathcal{V}_2}$. There are $3^9 = 19683$ possible $\{a, b\}$-reduced topology vectors of a sextet containing a, b. By ignoring $\{a, b\}$-reduced topology vectors in $\mathcal{V}_2$, there are 19668 possible branching vectors as well as 19668 branching numbers left. Actually, there are only 141 different branching numbers among these 19668 ones (this can be easily checked by a small program). Table 2.2 lists part of the branching vectors and the corresponding branching numbers (refer to Appendix D for the 141 different branching numbers). By examining these branching numbers, we obtain that the branching number is between 2.0161 and 2.0162 in the worst case. Thus the size of $T_2(k)$ is $O(2.0162^k)$.

Table 2.2: Some possible branching vectors and branching numbers of FPA-2SDMQI.

topology vector	**branching vector**	**branching number**
$(0,0,1,1,1,1,2,2,0)$	$(6,6,8,6,6,6,6,5,6,6,6,6,5,6,6)$	1.58005...
$(0,0,1,0,1,2,2,1,0)$	$(5,6,6,5,6,6,6,5,6,6,7,6,7,6,7)$	1.58142...
...	...	...
$(0,0,0,0,0,0,0,1,0)$	$(1,5,5,7,8,2,6,6,8,9,3,7,7,8,8)$	2.00904...
$(0,0,0,0,0,0,0,0,1)$	$(1,5,5,9,9,2,6,6,6,8,3,7,7,7,9)$	2.01615...

Similar to the procedure update in Sect. 2.3, the procedure update_2 runs in $O(n)$ time. In addition, building the list $\mathcal{C}_a$ costs $O(n^4)$ time. Hence the following theorem follows.

Theorem 2.7. *There exists an $O(2.0162^k n + n^4)$ fixed-parameter algorithm for the two-siblings-determined minimum quartet inconsistency problem.*

2.4.3 Solving the parameterized MQI problem by determining two siblings

Let T be an evolutionary tree on S such that Q is tree-like with T and Q differs from Q_T at most k quartet topologies. Note that every evolutionary tree with $|S| \geq 4$ leaves has at least two pairs of taxa which are siblings (Fig. 2.5 is an illustration for an evolutionary tree with only two pairs of siblings). Hence there must be two taxa which are siblings in T. So we devise another fixed-parameter algorithm, say FPA2-MQI, for the parameterized MQI problem.

Figure 2.5: An evolutionary tree with $n \geq 4$ leaves, where s_1, s_2 and s_{n-1}, s_n are two pairs of siblings.

First, the algorithm builds the list of unresolved quintets involving taxon s for every $s \in S$ and builds the list $\mathcal{V}_2$. Building these lists can be done in $O(n^5)$ time. And then Algorithm FPA2-MQI runs Algorithm FPA-2SDMQI for every two taxa, say a and b. Once there is an execution of Algorithm FPA-2SDMQI returning ACCEPT, then Algorithm FPA2-MQI returns ACCEPT, too. If such an evolutionary tree T exists, Algorithm FPA2-MQI must return ACCEPT. Thus the algorithm is valid. Therefore, by Theorem 2.7 we have an $O(2.0162^k n^3 + n^5)$ algorithm for the parameterized MQI problem. Here we summarize the above result into the following concluding theorem.

FPA2-MQI(Q, k)
/* Q: a complete set of quartet topologies; k: an integer parameter. */
begin
for every taxon $s \in S$ **do**
  build the list $\mathcal{C}_s$;
end for
$k^* \leftarrow k$;
for every two distinct taxa $a, b \in S$ **do**
  FPA-2SDMQI($Q, k^*, \mathcal{C}_a, a, b$);
  restoring Q and $\mathcal{C}_a$, and $k \leftarrow k^*$;
end for
end

Algorithm 2.4: FPA2-MQI: an $O(2.0162^k n^3 + n^5)$ algorithm for the parameterized MQI problem.

Theorem 2.8. *There exists an $O(2.0162^k n^3 + n^5)$ fixed-parameter algorithm for the parameterized minimum quartet inconsistency problem.*

2.5 An $O^*((1+\varepsilon)^k)$ Fixed-Parameter Algorithm

2.5.1 The algorithm

At the beginning of this section, let us consider some additional preliminaries. Let T denote an evolutionary tree on S such that Q_T differs from Q in at most k quartet topologies. For an integer $m \geq 2$, we say that taxa $a_1, \ldots, a_m$ are *adjacent* if there exists an edge $\mathbf{e} = (w, v)$ on T such that cutting $\mathbf{e}$ will produce a bipartition $(\{a_1, \ldots, a_m\}, S \setminus \{a_1, \ldots, a_m\})$ of S. In Fig. 2.6, cutting the edge $\mathbf{e}$ will derive four adjacent taxa a_1, a_2, a_3, and a_4. In addition, after $\mathbf{e} = (w, v)$ is cut, two binary trees, which are rooted at w and v respectively, will be produced. Note that two taxa on T are adjacent if and only if they are siblings on T.

Lemma 2.3. *Given an evolutionary tree T and an integer $2 \leq \omega \leq n/2$, there exists a set of m adjacent taxa as leaves on T, where $\omega \leq m \leq 2\omega - 2$.*

Proof. If there exists ω adjacent taxa on T, the lemma holds. Otherwise, assume that there is no subtree of T which has exactly m taxa as leaves. Let $T(s)$ denote the subtree of T which is rooted at a tree node s. There must exist some edge $\mathbf{e}^* = (w, v)$ such that cutting $\mathbf{e}^*$ will produce a bipartition $(A, S \setminus A)$, where $|A| > \omega$, $T(v)$ has A as its leaf set and two subtrees of $T(v)$ have both less than ω taxa as their leaves (otherwise, assume that t is one child of v such that $T(t)$ has more than ω taxa as leaves. Then we can recursively find a subtree of $T(t)$ rooted at some tree node x descendant of t until both two subtrees of $T(x)$ have less than ω taxa as their leaves). Assume that v has two children u and t, and $T(u)$ and $T(t)$ have p and p' taxa as leaves respectively, where $p, p' < \omega$. Since $|A| > \omega$, we have $p + p' > \omega$. Furthermore, $p + p' \leq 2\omega - 2$ since p and p' are both less than ω. So we have $\omega + 1 \leq p + p' \leq 2\omega - 2$. Therefore the lemma follows. □

Recall that Algorithm FPA2-MQI copes with siblings on an evolutionary tree first. In this section, we extend the idea of Algorithm FPA2-MQI to consider $m \geq 3$ adjacent taxa. We obtain another fixed-parameter algorithm called FPA3-MQI with two subroutines Algorithm MAKE-ADJ and Algorithm ADJ-Resolve. Assume that $A_m = \{a_1, \ldots, a_m\}$ is a set of adjacent taxa on T. In the following we introduce the

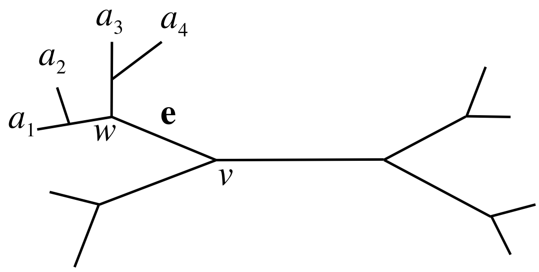

Figure 2.6: An evolutionary tree with adjacent taxa a_1, a_2, a_3, a_4.

main concepts of Algorithm FPA3-MQI, while the correctness will be clarified at the end of this subsection.

$(2, 2)$**-cleaning.** For every two taxa $a_i, a_j \in A_m$ and every two taxa $u, v \in S \setminus A_m$, we modify the topology of $\{a_i, a_j, u, v\}$ to be $[a_i a_j | uv]$. We call this part of the algorithm $(2, 2)$-*cleaning.*

$(3, 1)$**-cleaning.** Assume the parameter is k'. For $a_h, a_i, a_j \in A_m$ and $s \in S \setminus A_m$, without loss of generality we denote the type of quartet topology $[a_h a_i | a_j s]$ by 0, $[a_h a_j | a_i s]$ by 1, and $[a_h s | a_i a_j]$ by 2. We construct a set of all possible evolutionary trees $\mathcal{T}_{m+1}$ on the taxa in $A_m \cup \{x\}$, where x is an arbitrary taxon in $S \setminus A_m$, such that each $T' \in \mathcal{T}_{m+1}$ has at most k' different induced quartet topologies from Q. Afterwards, for each $T' \in \mathcal{T}_{m+1}$, we change the type of topology of every quartet $\{a_h, a_i, a_j, s\}$ into the same type of topology as $\{a_h, a_i, a_j, x\}$ has on T'. We call this part of the algorithm $(3, 1)$-*cleaning.*

$(1, 3)$**-cleaning.** Without loss of generality, we denote the type of quartet topology $[a_i w | xy]$ by 0, $[a_i x | wy]$ by 1, and $[a_i y | wx]$ by 2 for $a_i \in A_m$ and $w, x, y \in S \setminus A_m$. We build a list B_m of sets of three taxa $\{w, x, y\} \subseteq S \setminus A_m$ such that the topologies of $\{a_i, w, x, y\}$ are not all the same for $i = 1, \ldots, m$. Then we make all these quartet topologies be the same type by Algorithm MAKE-ADJ, which recursively branches on three possible types of these quartet topologies. We call this part of the algorithm $(1, 3)$-*cleaning.*

FPA3-MQI$(Q, k, \mathcal{C}_{a_1}, m)$
/* Q: a complete set of quartet topologies; k: an integer parameter;
$\mathcal{C}_{a_1}$: a list of unresolved quintets; m: an arbitrary integer. */
begin
$Q^* \leftarrow Q$; $\mathcal{C}^*_{a_1} \leftarrow \mathcal{C}_{a_1}$; $k^* \leftarrow k$;
for every set of m taxa $A_m = \{a_1, \ldots, a_m\} \subseteq S$ **do**
  for every two taxa $a_i, a_j \in A_m$ and every two taxa $u, v \in S \setminus A_m$ **do**
    if $k^* \leq 0$ **then**
      return
    else
      change the quartet topology of $\{a_i, a_j, u, v\}$ in Q^* to be $[a_i a_j | uv]$ if $[a_i a_j | uv] \notin Q^*$, and then update $\mathcal{C}^*_{a_1}$ and $k^* \leftarrow k^* - 1$;
    end if
  end for
  build a set of all possible evolutionary trees $\mathcal{T}_{m+1}$ such that each $T' \in \mathcal{T}_{m+1}$ is an evolutionary tree on $A_m \cup \{x\}$, where x is an arbitrary taxon in $S \setminus A_m$ and $|Q_{T'} \setminus Q^*| \leq k^*$;
  build a list B_m of sets of three taxa $w, x, y \in S \setminus A_m$ such that topologies of $\{a_i, w, x, y\}$ in Q^* are not all the same for all $1 \leq i \leq m$;
  $Q^{**} \leftarrow Q^*$; $\mathcal{C}^{**}_{a_1} \leftarrow \mathcal{C}^*_{a_1}$; $k^{**} \leftarrow k^*$;
  if $\mathcal{T}_{m+1} = \emptyset$ **then**
    return
  else
    for each $T' \in \mathcal{T}_{m+1}$ **do**
      $k^{**} \leftarrow k^{**} - |Q_{T'} \setminus Q^{**}|$;
      change the quartet topologies in Q^{**} over A_m to those in $Q_{T'}$;
      for every taxon $s \in S \setminus A_m$ and every three taxa $a_h, a_i, a_j \in A_m$, change the topology of $\{a_h, a_i, a_j, s\}$ to the one of the same type as $\{a_h, a_i, a_j, x\}$ has; update $\mathcal{C}^{**}_{a_1}$;
      if MAKE-ADJ$(Q^{**}, \mathcal{C}^{**}_{a_1}, k^{**})$ returns ACCEPT **then**
        return ACCEPT;
      else
        restore $(Q^{**}, \mathcal{C}^{**}_{a_1})$ to $(Q^*, \mathcal{C}^*_{a_1})$, and $k^{**} \leftarrow k^*$;
      end if
    end for
  end if
  restore $(Q^*, \mathcal{C}^*_{a_1})$ to $(Q, \mathcal{C}_{a_1})$, delete B_m, and $k^* \leftarrow k$;
end for
end

Algorithm 2.5: FPA3-MQI: an $O^*((1+\varepsilon)^k)$ algorithm for the parameterized MQI problem.

MAKE-ADJ($Q, \mathcal{C}_{a_1}, k$)
/* Q: a complete set of quartet topologies; $\mathcal{C}_{a_1}$: a list of unresolved quintets;
k: an integer parameter. */
begin
if $\mathcal{C}_{a_1}$ is empty and $k \geq 0$ **then**
    return ACCEPT;
else if $k \leq 0$ **then**
    return
end if
while $B_m \neq \emptyset$ **do**
    extract $\{w, x, y\}$ from B_m;
    for each type $i \in \{0, 1, 2\}$ **do**
        change all the topologies of $\{a_1, w, x, y\}, \ldots, \{a_m, w, x, y\}$ to topologies of type i; let $Q', \mathcal{C}'_{a_1}, k'$ be the changed $Q, \mathcal{C}_{a_1}, k$ respectively;
        MAKE-ADJ($Q', \mathcal{C}'_{a_1}, k'$);
    end for
end while
if ADJ-Resolve($Q, k, \mathcal{C}_{a_1}$) returns ACCEPT **then**
    return ACCEPT;
end if
end

Algorithm 2.6: MAKE-ADJ: a subroutine of FPA3-MQI.

Quintet cleaning. After $(2,2)$-cleaning, $(3,1)$-cleaning and $(1,3)$-cleaning, assume that the parameter is k'' for the moment. We try to resolve all the unresolved quintets in $\mathcal{C}_{a_1}$ through Algorithm ADJ-Resolve, which changes at most k'' quartet topologies in Q. We call this part of the algorithm *quintet cleaning*.

Lemma 2.4. *Assume that $A_m = \{a_1, \ldots, a_m\}$ and the list of unresolved quintet is $\mathcal{C}_{a_1}$, then after $(2,2)$-cleaning, $(3,1)$-cleaning, and $(1,3)$-cleaning, $\mathbf{q} \cap A_m = \{a_1\}$ for every $\mathbf{q} \in \mathcal{C}_{a_1}$.*

Proof. We prove this lemma by contradiction as follows. Assume that $A_m = \{a_1, \ldots, a_m\}$, $\mathcal{C}_{a_1}$ is the list of unresolved quintet considered for the moment, and $(2,2)$-cleaning, $(3,1)$-cleaning, and $(1,3)$-cleaning are done. For an unresolved quintet $\mathbf{q} \in \mathcal{C}_{a_1}$, we consider four cases for the proof: $|(\mathbf{q} \cap A_m) \setminus \{a_1\}| = i$, where $i = 1, 2, 3, 4$. First, without loss of generality, assume that $\mathbf{q} = \{a_1, a_2, w, x, y\}$, where $a_1, a_2 \in A_m$ and $w, x, y \in S \setminus A_m$. In this quintet, the quartets $\{a_1, a_2, w, x\}$, $\{a_1, a_2, w, y\}$, and $\{a_1, a_2, x, y\}$ have topologies $[a_1a_2|wx]$, $[a_1a_2|wy]$, and $[a_1a_2|xy]$

respectively, due to $(2,2)$-cleaning of the Algorithm FPA3-MQI. By $(1,3)$-cleaning of the Algorithm, the quartets $\{a_1, w, x, y\}$ and $\{a_2, w, x, y\}$ have the same type of topologies. Let us consider Fig. 2.7 for an illustration. If $[a_1w|xy] \in Q$, then the quintet has the topology in (a) of Fig. 2.7. Similarly, we can derive the other two quintet topologies in (b) and (c) of Fig. 2.7, so the quintet $\{a_1, a_2, w, x, y\}$ must be resolved. Then a contradiction occurs.

```
ADJ-Resolve(Q, k, C_{a1})
/* Q: a complete set of quartet topologies; k: an integer parameter;
   C_{a1}: a list of unresolved quintets. */
begin
  if C_{a1} is empty and k ≥ 0 then
      return ACCEPT;
  else if k ≤ 0 then
      return
  end if
  extract an unresolved quintet q from C_{a1};
  for each μ ∈ V do
      (Q', C'_{a1}, k') ← update_m(Q, C_{a1}, q, μ, k);
      ADJ-Resolve(Q', k', C'_{a1});
  end for
end
```

Algorithm 2.7: ADJ-Resolve: a subroutine of MAKE-ADJ.

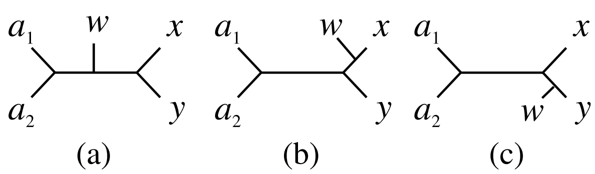

Figure 2.7: Possible topologies for the quintet $\{a_1, a_2, w, x, y\}$.

Second, without loss of generality we assume that $\mathbf{q} = \{a_1, a_2, a_3, x, y\}$, where $a_1, a_2, a_3 \in A_m$ and $x, y \in S \setminus A_m$. In this quintet, the quartets $\{a_1, a_2, x, y\}$, $\{a_1, a_3, x, y\}$, and $\{a_2, a_3, x, y\}$ have topologies $[a_1a_2|xy]$, $[a_1a_3|xy]$, and $[a_2a_3|xy]$ respectively, due to $(2,2)$-cleaning of the algorithm. Thus there are three possible quintet topologies for this quintet. Recall that $\{a_1, a_2, a_3, x\}$ and $\{a_1, a_2, a_3, y\}$ have the same type of quartet topologies due to $(3,1)$-cleaning of the algorithm. If $[a_1a_2|a_3x] \in Q$, then $[a_1a_2|a_3y] \in Q$ and hence we have a topology for the quintet in

(a) of Fig. 2.8. Similarly for the other possible quartet topologies of $\{a_1, a_2, a_3, x\}$, we obtain the other two quintet topologies in (b) and (c) of Fig. 2.8. So the quintet $\{a_1, a_2, a_3, x, y\}$ must be also resolved. Then a contradiction occurs.

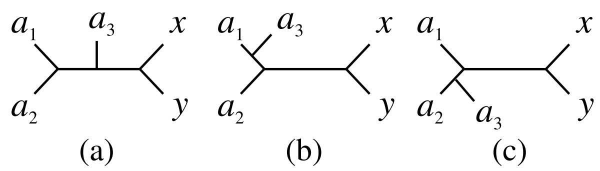

Figure 2.8: Possible topologies for the quintet $\{a_1, a_2, a_3, x, y\}$.

Third, without loss of generality we assume $\mathbf{q} = \{a_1, a_2, a_3, a_4, s\}$, where $a_1, a_2, a_3, a_4 \in A_m$ and $s \in S \setminus A_m$. Recall that for some fixed taxa $x \in S \setminus A_m$, the tree topology of $A_m \cup \{x\}$ is determined because of $(3, 1)$-cleaning of the algorithm. Moreover, all the quartets in $\{a_1, a_2, a_3, a_4, s\}$ have the same type of quartet topologies as $\{a_1, a_2, a_3, a_4, x\}$ have. So the quintet $\{a_1, a_2, a_3, a_4, s\}$ must be resolved. Then a contradiction occurs again. As to the fourth case of the proof, i.e., the quintets involving five taxa in A_m, their topologies are also determined by $(3, 1)$-cleaning of the algorithm, so they must be resolved. Therefore, we have shown that as long as $(2, 2)$-cleaning, $(3, 1)$-cleaning, and $(1, 3)$-cleaning of the algorithm are done, there is no unresolved quintet in $\mathcal{C}_{a_1}$ containing taxa in A_m except a_1. Hence the lemma follows. $\square$

Note that there do not always exist ω adjacent taxa in an evolutionary tree for an arbitrary integer ω. By Lemma 2.3, we know there must be m taxa which are adjacent in an evolutionary tree, where $\omega \leq m \leq 2\omega - 2$. Assume that we are given an integer ω as an additional input. Then to solve the parameterized MQI problem, first we build a list of unresolved quintet involving s for each $s \in S$, then we run Algorithm FPA3-MQI for every $m \in \{\omega, \dots, 2\omega - 2\}$.

By Lemma 2.4 we know that each unresolved quintet $\mathbf{q} \in \mathcal{C}_{a_1}$ contains a_1 and the other four taxa from $S \setminus A_m$. The procedure update_m is similar to the procedure update in Sect. 2.3. Yet if a quartet topology of $\{a_1, w, x, y\}$, where $w, x, y \in \mathbf{q} \setminus a_1$, is changed, the procedure not only changes quartet topologies according to μ, but also changes the topologies of $\{a_2, w, x, y\}, \{a_3, w, x, y\}, \dots, \{a_m, w, x, y\}$ together into the same type as $\{a_1, w, x, y\}$ has. Let d denote the number of quartet topologies changed by update_m. Then the procedure updates the set $\mathcal{C}_{a_1}$ and the parameter k to be $\mathcal{C}'_{a_1}$ and k' respectively, where k' is $k-d$.

Correctness.

Recall that we use T to denote an evolutionary tree on S such that Q_T differs from Q in at most k quartet topologies. Given an arbitrary integer $2 \le \omega \le n/2$, there exists m adjacent taxa in T, where $\omega \le m \le 2\omega - 2$. So we can assume that there is a set of adjacent taxa $A_m = \{a_1, \ldots, a_m\} \subseteq S$ on T. Since the taxa in A_m are adjacent, the path connecting every two taxa $a_i, a_j \in A_m$ and the path connecting two taxa $u, v \in S \backslash A_m$ will be disjoint and hence the topology of $\{a_i, a_j, u, v\}$ must be $[a_i a_j | uv]$. So $(2,2)$-cleaning is valid. In addition, once the topology of $\{a_h, a_i, a_j, x\}$ is fixed for $a_h, a_i, a_j \in A_m$ and some $x \in S \setminus A_m$, the quartets $\{a_h, a_i, a_j, s\}$ must have the same type of quartet topologies as $\{a_h, a_i, a_j, x\}$ has one T. Hence $(3,1)$-cleaning is valid. Furthermore, the path structure connecting a_i, w, x, y on T must be the same for all $i \in \{1, \ldots, m\}$ and every three taxa $w, x, y \in S \setminus A_m$, so $(1,3)$-cleaning is valid. After $(2,2)$-cleaning, $(3,1)$-cleaning and $(1,3)$-cleaning, there are only unresolved quintets involving a_1 by Lemma 2.4. Thus Algorithm ADJ-Resolve together with the procedure update_m is valid for quintet cleaning. The number of unresolved quintets in $\mathcal{C}_{a_1}$ can be always decreased until Q is tree-like. The algorithm returns ACCEPT only when $\mathcal{C}_{a_1}$ is empty and no more than k quartet topologies are changed. Therefore by Theorem 2.4 the Algorithm is correct.

2.5.2 Time complexity

Nonrecursive steps.

Building and updating the lists of unresolved quintets. It is clear that building $\mathcal{C}_s$ for every $s \in S$ costs $O(n^5)$ time. For a fixed $A_m = \{a_1, a_2, \ldots, a_m\}$, the algorithm considers only $\mathcal{C}_{a_1}$. Whenever a quartet topology is changed, only $O(n)$ quintets will be examined in order to update $\mathcal{C}_{a_1}$, so updating the list $\mathcal{C}_{a_1}$ costs $O(n)$ time for each time.

$(2,2)$**-cleaning and** $(3,1)$**-cleaning.** There are at most $O(\binom{m}{2} \cdot \binom{n-2}{2}) = O(m^2 n^2)$ quartets examined by $(2,2)$-cleaning, so it costs $O(m^2 n^3)$ time for $(2,2)$-cleaning (the additional n factor here as well as the rest analysis in this paragraph comes from updating a list of unresolved quintets). As to $(3,1)$-cleaning, constructing $\mathcal{T}_{m+1}$ costs $O(h(m))$ time, where $h(m)$ depends on m only. After $\mathcal{T}_{m+1}$ is constructed, for each $T' \in \mathcal{T}_{m+1}$, first the algorithm spends $O(\binom{m}{4} \cdot n) = O(m^4 n)$ time to change the quartet topologies in Q over A_m to those in $Q_{T'}$. Second, the algorithm spends $O(\binom{m}{3} \cdot (n-m-1) \cdot n) = O(m^3 n^2)$ time to make every quartet $\{a_h, a_i, a_j, s\}$ have the same topology as $\{a_h, a_i, a_j, x\}$ has, where $a_h, a_i, a_j \in A_m$, $s, x \in S \setminus A_m$, while x is a leaf on T'.

The procedure update_m**.** Assume that the list of unresolved quintets is $\mathcal{C}_{a_1}$ for the moment. It is clear that making an unresolved quintet in $\mathcal{C}_{a_1}$ resolved and then updating $\mathcal{C}_{a_1}$ cost $O(n)$ time. Moreover, as long as the topology of a quartet $\{a_1, w, x, y\}$, where $w, x, y \in S \backslash A_m$, is changed, the procedure changes the topologies of $\{a_i, w, x, y\}$ for $i = 2, 3, \ldots, m$. Thus the procedure update_m runs in $O(mn)$ time.

Recursive steps.

$(1,3)$**-cleaning by the recursive algorithm** MAKE-ADJ**.** The preprocessing for $(1,3)$-cleaning builds a list B_m (Line 11 of Algorithm FPA3-MQI), which costs $O(\binom{n-m}{3} m) = O(mn^3)$ time. Then let us consider the quartets $\{a_1, w, x, y\}$, ..., $\{a_m, w, x, y\}$, where $w, x, y \in S \backslash A_m$. Without loss of generality, we denote the quartet topologies $[a_i w|xy]$, $[a_i x|wy]$, and $[a_i y|wx]$ to be type 0, 1, and 2 respectively, for all $i = 1, \ldots, m$. Let m_j be the number of quartets in $\{\{a_1, w, x, y\}, \ldots, \{a_m, w, x, y\}\}$ which have topologies of type j. It is clear that $m_0 + m_1 + m_2 = m$. $(1,3)$-cleaning branches on these three types to make every quartet $\{a_i, w, x, y\}$, where $a_i \in A_m$, have the same type of topology. Then $(1,3)$-cleaning of the algorithm has a recurrence of $T(k) = T(k-(m_1+m_2)) + T(k-(m_0+m_2)) + T(k-(m_0+m_1))$. So we have a branching vector $(m_1 + m_2, m_0 + m_2, m_0 + m_1)$. Let $r_0 = m_1 + m_2$, $r_1 = m_1 + m_2$ and $r_2 = m_0 + m_1$. Since the order of a branching vector does not change its branching number, without loss of generality we assume that $0 < r_0 \le r_1 \le r_2 \le m$. Since $m_0 + m_1 + m_2 = m$, we have $r_0 + r_1 + r_2 = 2m$ and $r_1, r_2 \ge m/2$. The next lemma shows that the size of the depth-bounded search tree of $(1,3)$-cleaning is $O((1 + 5m^{-1/4})^k)$. Moreover, it can be proved to be $O((1 + 2m^{-1/2})^k)$ if $m \ge 19$.

Lemma 2.5. *Given a branching vector* (r_0, r_1, r_2)*, where* $0 < r_0 \le r_1 \le r_2 \le m$*,* $r_0+r_1+r_2 = 2m$ *and* $r_1, r_2 \ge m/2$*, then we have a branching number* $\alpha < 1+5m^{-1/4}$*. Furthermore,* $\alpha < 1 + 2m^{-1/2}$ *if* $m \ge 19$*.*

Proof. The reflected characteristic polynomial of (r_0, r_1, r_2) is $1 - z^{r_0} - z^{r_1} - z^{r_2}$. Let $f(z) = 1 - z^{r_0} - z^{r_1} - z^{r_2}$. We have $f(0) = 1$ and $f(1) = -2$, so there is a root of $f(z)$ in $[0, 1]$. The derivative $f'(z) = -r_0 z^{r_0-1} - r_1 z^{r_1-1} - r_2 z^{r_2-1}$. We can derive that $f(z)$ is monotonically decreasing in $[0, 1]$ since $f'(z) \le 0$ for $0 \le z \le 1$. Let us define $g(z) = 1 - z - 2z^{m/2}$. Similarly, $g(z)$ has a root in $[0, 1]$ and is monotonically decreasing in $[0, 1]$. Since $z^{r_0} \le z$ and $z^{r_1}, z^{r_2} \le z^{m/2}$, we have $g(z) \le f(z)$. We can then derive that there is a root of $g(z)$ which is smaller than the root of $f(z)$.

Let $0 \le z_0 \le 1$ be a root of $g(z)$, i.e., $g(z_0) = 0$. Let $z_1 = 1 - m^{-1/4}$ and $z_2 = 1 - m^{-1/2}$, so $0 \le z_1, z_2 \le 1$. Then we have $g(z_1) = m^{-1/4} - 2(1 - m^{-1/4})^{m/2} > m^{-1/4} - 2e^{-m^{3/4}/2}$, and $g(z_2) = m^{-1/2} - 2(1 - m^{-1/2})^{m/2} > m^{-1/2} - 2e^{-m^{1/2}/2}$. So $g(z_1) > 0$ when $m \ge 3$ and $g(z_2) > 0$ when $m \ge 19$. If $g(z_1) > 0$, then z_0 must be bigger than z_1 because $g(z)$ is monotonically decreasing in $[0, 1]$. So we have $z_0 > 1 - m^{-1/4}$ for $m \ge 3$. Similarly, we have $z_0 > 1 - m^{-1/2}$ if $m \ge 19$. Therefore, the branching number α is smaller than $1/(1 - m^{-1/4}) < 1 + 5m^{-1/4}$. Furthermore, if $m \ge 19$, α is smaller than $1/(1 - m^{-1/2}) < 1 + 2m^{-1/2}$. The lemma is then proved. □

Quintet cleaning by the recursive algorithm ADJ-Resolve. Assume that the list of unresolved quintets is $\mathcal{C}_{a_1}$. Let $\mathbf{q} = \{a_1, w, x, y, z\}$ be an unresolved quintet in $\mathcal{C}_{a_1}$, and let $\mathbf{v_q} = (\mathbf{v_q}(1), \mathbf{v_q}(2), \mathbf{v_q}(3), \mathbf{v_q}(4), \mathbf{v_q}(5))$ denote the topology vector of $\mathbf{q}$, where $\mathbf{v_q}(1)$, $\mathbf{v_q}(2)$, $\mathbf{v_q}(3)$, $\mathbf{v_q}(4)$, and $\mathbf{v_q}(5)$ are the types of topologies of $\{a_1, w, x, y\}$, $\{a_1, w, x, z\}$, $\{a_1, w, y, z\}$, $\{a_1, x, y, z\}$, and $\{w, x, y, z\}$ respectively, with respect to Q. Recall that $\mathcal{V} = \{\mu_1, \ldots, \mu_{15}\}$ is a set of topology vectors of 15 possible quintet topologies for a quintet, such that each $\mu_i = (\mu_i(1), \mu_i(2), \mu_i(3), \mu_i(4), \mu_i(5)) \in \mathcal{V}$ stands for the ith topology vector in $\mathcal{V}$. If $\mathbf{q}$ is resolved, there exists exactly one $\mu_i \in \mathcal{V}$ such that $\mathbf{v_q}(j) = \mu_i(j)$ for each $1 \le j \le 5$. Let $\mathbf{v_q}(j) \oplus \mu_i(j)$ denote whether $\mathbf{v_q}(j)$ and $\mu_i(j)$ are different. That is, for $1 \le j \le 5$ we denote $\mathbf{v_q}(j) \oplus \mu_i(j) = 1$ if $\mathbf{v_q}(j) \ne \mu_i(j)$ and $\mathbf{v_q}(j) \oplus \mu_i(j) = 0$ otherwise.

For an unresolved quintet $\mathbf{q}$, let $\mathbf{b}(\mathbf{q})$ denote the branching vector of the recurrence of the quintet cleaning for $\mathbf{q}$. By the descriptions of quintet cleaning and the procedure update_m of the algorithm, we derive that $\mathbf{b}(\mathbf{q}) = (\mathbf{b}_1(\mathbf{q}), \ldots, \mathbf{b}_{15}(\mathbf{q}))$, where $\mathbf{b}_i(\mathbf{q}) = m\left(\sum_{j=1}^{4} \mathbf{v_q}(j) \oplus \mu_i(j)\right) + \mathbf{v_q}(5) \oplus \mu_i(5)$. Note that $\mathbf{b}_i(\mathbf{q}) \neq 0$ since $\mathbf{q}$ is unresolved. Let us consider the following lemma.

Lemma 2.6. *Given an unresolved quintet* $\mathbf{q}$ *in* $\mathcal{C}_{a_1}$. *If* $\mathbf{b}_i(\mathbf{q}) = 1$ *for some* $1 \leq i \leq 15$*, then* $\mathbf{b}_h(\mathbf{q}) = cm$ *for each* $1 \leq h \leq 15$ *except* i*, where* $1 \leq c \leq 4$.

Proof. If $\mathbf{b}_i(\mathbf{q}) = 1$ for some $1 \leq i \leq 15$, then we have $\mathbf{v_q}(j) \oplus \mu_i(j) = 0$ for $1 \leq j \leq 4$ and $\mathbf{v_q}(5) \oplus \mu_i(5) = 1$. By observing the topology vectors in $\mathcal{V}$, we obtain that $(\mu_{i'}(1), \mu_{i'}(2), \mu_{i'}(3), \mu_{i'}(4))$ and $(\mu_{i''}(1), \mu_{i''}(2), \mu_{i''}(3), \mu_{i''}(4))$ are different, for every two $\mu_{i'}, \mu_{i''} \in \mathcal{V}$. Thus for every $h \in \{1, \ldots, m\} \backslash \{i\}$, we have $\sum_{j=1}^{4} \mathbf{v_q}(j) \oplus \mu_h(j) = c$, where $1 \leq c \leq 4$. Therefore the lemma follows. $\square$

Let β denote the branching number corresponding to $\mathbf{b}(\mathbf{q})$. Since changing the order of the branching vector does not affect its branching number, without loss of generality we assume that $\mathbf{b}(\mathbf{q}) = (\mathbf{b}_1(\mathbf{q}), \ldots, \mathbf{b}_{15}(\mathbf{q}))$, where $\mathbf{b}_1(\mathbf{q}) \leq \mathbf{b}_2(\mathbf{q}) \leq \ldots, \mathbf{b}_{15}(\mathbf{q})$. By Lemma 2.6 and Theorem 2.2, we obtain that the branching number β is no bigger than that of $(1, m_1, m_2, \ldots, m_{14})$, where $m_1 = m_2 = \ldots = m_{14} = m$. Thus the size of the depth-bounded search tree of Algorithm $\mathsf{ADJ\text{-}Resolve}$ (i.e., quintet cleaning) is $O(\gamma^k)$, where γ is the branching number of $(1, m_1, m_2, \ldots, m_{14})$. Lemma 2.7 shows that γ is less than $1 + 12m^{-1/12}$. Furthermore, it can be proved to be $O((1 + 2m^{-1/2})^k)$ if $m \geq 17$.

Lemma 2.7. *Given a branching vector* $(1, m_1, m_2, \ldots, m_{14})$*, where* $m_i = m$ *for each* $1 \leq i \leq 14$*, then we have a branching number* $\gamma < 1 + 12m^{-1/12}$*. Furthermore,* $\gamma < 1 + 2m^{-1/2}$ *if* $m \geq 17$.

Proof. The reflected characteristic polynomial of $(1, m_1, m_2, \ldots, m_{14})$ is $1 - z - 14z^m$. Let $f(z) = 1 - z - 14z^m$. We have $f(0) = 1$ and $f(1) = -1$, so there is a root of $f(z)$ in $[0, 1]$. The derivative of $f(z)$ is $f'(z) = -1 - 14mz^{m-1}$, so it is obvious that $f(z)$ is monotonically decreasing. Let $0 \leq z_0 \leq 1$ be the root of $f(z)$. Let $z_1 = 1 - m^{-1/12}$ and $z_2 = 1 - m^{-1/2}$, so $0 \leq z_1, z_2 \leq 1$. Then we have $f(z_1) = m^{-1/12} - 14(1 - m^{-1/12})^m > m^{-1/12} - 14e^{-m^{11/12}}$, and $f(z_2) = m^{-1/2} - 14(1 - m^{-1/2})^m > m^{-1/2} - 14e^{-m^{1/2}}$. Hence $f(z_1) \geq 0$ for $m \geq 3$ and

$f(z_2) \geq 0$ for $m \geq 17$. Note that $z_0 \geq z_1$ and $z_0 \geq z_2$ if $f(z_1) > 0$ and $f(z_2) > 0$ since $f(z)$ is monotonically decreasing in $[0,1]$. Then we have $z_0 > 1 - m^{-1/12}$ for $m \geq 3$ and $z_0 > 1 - m^{-1/2}$ for $m \geq 17$. Therefore, the branching number γ is smaller than $1/(1-m^{-1/12}) < 1 + 12m^{-1/12}$. Furthermore, γ is smaller than $1/(1-m^{-1/2}) < 1 + 2m^{-1/2}$ if $m \geq 17$. □

Overall time complexity.

Since each leaf node of the depth-bounded search tree of $(1,3)$-cleaning is a root node of the depth-bounded search tree of quintet cleaning, by the analysis in the previous subsection, we obtain that the size of the depth-bounded search tree the algorithm in the worst case is $O((1+2m^{-1/2})^k)$, for large enough $m \geq 19$. When a set of m adjacent taxa A_m is given, since it costs $O(mn)$ time at each node in the search tree, the time complexity for the search tree is $O((1+2m^{-1/2})^k mn)$. Assume that $1 + 2m^{-1/2} \leq 1 + \varepsilon$ for some constant $\varepsilon > 0$. We obtain $m \geq (2/\varepsilon)^2$. Thus after the lists of unresolved quintets $\{\mathcal{C}_s \mid s \in S\}$ are built, we run Algorithm FPA3-MQI for every $(2/\varepsilon)^2 \leq m \leq 2(2/\varepsilon)^2 - 2$ and every set of m taxa in S. Let ω denote $(2/\varepsilon)^2$. By the analysis in the previous subsection, we obtain the overall time complexity of the algorithm as follows.

$$
\begin{aligned}
& O\left(n^5 + \sum_{m=\omega}^{2\omega-2} \binom{n}{m} \left(m^2n^3 + mn^3 + h(m) \cdot \left(m^4n + m^3n^2 + (1+\varepsilon)^k mn\right)\right)\right) \\
= \;& O\big(n^5 + (\omega-1)n^{2\omega-2}\big(4\omega^2n^3 + 2\omega n^3 + h(2\omega) \cdot (16\omega^4 n + 8\omega^3n^2 + 2(1+\varepsilon)^k \omega n)\big)\big) \\
= \;& O((1+\varepsilon)^k n^{2\omega-1} + n^{2\omega+1} + n^5) \\
= \;& O((1+\varepsilon)^k n^{8/\varepsilon^2-1} + n^{8/\varepsilon^2+1} + n^5).
\end{aligned}
$$

Consider the first line of above deduction. Recall that the term n^5 comes from building C_s for $s \in S$. The summation and the term $\binom{n}{m}$ arise due to exhaustively taking all the possibilities of A_m (i.e., the set of m adjacent taxa) into consideration. The term m^2n^3 comes from $(2,2)$-cleaning. The term mn^3 comes from the preprocessing of $(1,3)$-cleaning and quintet-cleaning. The term $h(m)$ arises from the construction of all possible evolutionary trees on $A_m \cup \{x\}$, where x is a taxon not in A_m. The terms m^4n and m^3n^2 arise from $(3,1)$-cleaning. The rest term $(1+\varepsilon)^k mn$ in the first line is derived from the analysis of the size of depth-bounded search tree of $(1,3)$-cleaning and quintet-cleaning. The second equality holds since $m < 2\omega - 2 < 2\omega$ and $\binom{n}{m} = O(n^m)$. Therefore we have an $O^*((1+\varepsilon)^k)$ fixed-parameter algorithm for the parameterized MQI problem. Hence the following concluding theorem follows.

Theorem 2.9. *There exists an $O^*((1+\varepsilon)^k)$ time fixed-parameter algorithm for the parameterized minimum quartet inconsistency problem, where $\varepsilon > 0$ is an arbitrarily small constant and the degree of the involved polynomial in the running time has dependence on ε.*

Chapter 3

A Property Tester for Tree-Likeness of Quartet Topologies

As shown in the previous chapter, there are efficient fixed-parameter algorithms for the parameterized MQI problem. They are believed to work well when the parameter k is small. However, when k gets much bigger, our fixed-parameter algorithms are no more efficient. In particular, say $k = c \cdot \binom{n}{4}$, by applying any of our fixed-parameter algorithms for the parameterized MQI problem we can only derive that the problem can be solved in $2^{O(n^4)}$ time. This leads us to consider the notion of property testing for this circumstance.

In this chapter, we focus on the task of testing whether the a complete set of quartet topologies Q is tree-like. Firstly, in Sect. 3.1 we define the setting of property testing for this property. In Sect. 3.2 we prove that there exists a complete set of quartet topologies Q that has at least $\Omega(n^4)$ quartet errors, hence it is possible for Q to be ϵ-far from being tree-like. Then, in Sect. 3.3, we present a non-adaptive $O(n^3/\epsilon)$ property tester with one-sided error for testing if a complete Q is tree-like. Such a property tester, say $\mathcal{M}$, fulfills the following conditions:

i. $\mathcal{M}$ answers "yes" with probability at least $2/3$ if Q is tree-like;

ii. $\mathcal{M}$ answers "no" with probability at least $2/3$ if Q is ϵ-far from being tree-like (i.e., Q is not tree-like unless at least $\epsilon\binom{n}{4}$ quartet topologies are changed).

We end this chapter with discussions for the case that Q is incomplete. We give some convincing evidences that our property tester seems unlikely to work for incomplete Q's due to the reason that local consistency of quintets does not guarantee the global consistency of the whole set of quartet topologies.

3.1 Preliminaries

Assume that Q is complete and $|S| \geq 5$. We regard Q as a function $f_Q : \{\{a,b,c,d\} \mid a,b,c,d \in S\} \mapsto \{0,1,2\}$, where $f_Q(\{a,b,c,d\})$ is equal to the type of the topology of $\{a,b,c,d\}$ in Q. The domain size of the function f_Q is then equal to $\binom{n}{4}$. By the above settings, a query of f_Q here retrieves the topology of a quartet in Q. We utilize an array of $\binom{n}{4}$ entries, where each entry stores the type of the topology of a quartet over S. For two complete sets of quartet topologies Q_1 and Q_2, let $\delta(Q_1, Q_2) = |Q_1 \setminus Q_2|/\binom{n}{4}$ denote the fraction of the $\binom{n}{4}$ quartets where Q_1 differs from Q_2. We define $\mathcal{P}_{tree}$ to be the set of all the functions f_{Q^*} where Q^* is tree-like. Define that $\Delta(Q, \mathcal{P}_{tree}) = \min_{Q^* \in \mathcal{P}_{tree}} \delta(Q, Q^*)$. Clearly, Q is tree-like if and only if $\Delta(Q, \mathcal{P}_{tree}) = 0$. We say that Q is *ϵ-far from being tree-like* if the error number of Q is at least $\epsilon\binom{n}{4}$, that is, $\Delta(Q, \mathcal{P}_{tree}) \geq \epsilon$.

Consider quintet topology (v) and quintet topology (x) in Fig. 2.3. Quintet topology (v) induces five quartet topologies $[ab|cd]$, $[ab|ce]$, $[ab|de]$,$[ac|de]$, and $[bc|de]$, while quintet topology (x) induces $[ac|bd]$, $[ac|be]$, $[ab|de]$, $[ac|de]$, and $[bc|de]$, so there are two quartets (i.e., $\{a,b,c,d\}$ and $\{a,b,c,e\}$) whose topologies induced by quintet topology (v) are different from those induced by quintet topology (x). By exhaustively observing the induced quartet topologies of each quintet topology in Fig 2.3, we can easily obtain the following fact.

Fact 3.1. Any two topologies of a quintet differ in at least two induced quartet topologies.

3.2 Existence of an Instance Far from Being Tree-Like

In this section, we show that there exists a complete set of quartet topologies that is at least 0.04-far from being tree-like, that is, its error number is at least $0.04\binom{n}{4}$. The sketch of the proof is as follows. First, we show that there exists a set of $\gamma\binom{n}{4}$ quintets $\mathcal{U}$ over S for some constant γ, such that every two quintets of $\mathcal{U}$ do not share any quartet. We present two ways for constructing such a set $\mathcal{U}$ and show that $\gamma \geq 0.04$. Second, by considering an arbitrary tree-like set Q^*, for each quintet $\mathbf{u} \in \mathcal{U}$ with respect to Q^*, we change one quartet topology of the subset quartets of $\mathbf{u}$ to make $\mathbf{u}$ unresolved. We show that the error number of the resulting set of quartet topologies is at least $0.04\binom{n}{4}$.

A simple construction of $\mathcal{U}$. Let us label the taxa in S by $S = \{s_1, s_2, \ldots, s_n\}$. Let $\mathcal{U}$ denote the set $\{\{s_{n/5+i_1}, s_{2n/5+i_2}, s_{3n/5+i_3}, s_{4n/5+i_4}, s_{i_1+i_2+i_3+i_4}\} \mid 1 \leq i_1, i_2, i_3, i_4 \leq n/20\}$. Clearly, the five taxa of every element of $\mathcal{U}$ are distinct, $\mathcal{U}$ is indeed a set of quintets over S. Moreover, each 4-tuple (i_1, i_2, i_3, i_4) corresponds to a quintet in $\mathcal{U}$, so the size of $\mathcal{U}$ is $(n/20)^4 = n^4/160000 > 0.0015\binom{n}{4}$.

Lemma 3.1. *Any two quintets in $\mathcal{U}$ do not share any quartet.*

Proof. Assume the contrary that two quintets $\mathbf{u}, \mathbf{v}$ in $\mathcal{U}$ share a quartet. Let $\mathbf{u} = \{s_{n/5+i_1}, s_{2n/5+i_2}, s_{3n/5+i_3}, s_{4n/5+i_4}, s_{i_1+i_2+i_3+i_4}\}$ and $\mathbf{v} = \{s_{n/5+j_1}, s_{2n/5+j_2}, s_{3n/5+j_3}, s_{4n/5+j_4}, s_{j_1+j_2+j_3+j_4}\}$ respectively, where $1 \leq i_1, \ldots, i_4, j_1, \ldots, j_4 \leq n/20$. If $\mathbf{u}$ and $\mathbf{v}$ share the quartet $\{s_{n/5+i_1}, s_{2n/5+i_2}, s_{3n/5+i_3}, s_{4n/5+i_4}\}$, that is, $i_1 = j_2, i_2 = j_2, i_3 = j_3, i_4 = j_4$, we have $i_1 + i_2 + i_3 + i_4 = j_1 + j_2 + j_3 + j_4$. Then $\mathbf{u}$ and $\mathbf{v}$ are actually the same quintets, so a contradiction occurs. As for the other possibilities that $\mathbf{u}$ and $\mathbf{v}$ share a quartet, without loss of generality, we assume that they share the quartet $\{s_{n/5+i_1}, s_{2n/5+i_2}, s_{3n/5+i_3}, s_{i_1+i_2+i_3+i_4}\}$. We obtain that $i_1 = j_1, i_2 = j_2, i_3 = j_3$ and $i_1 + i_2 + i_3 + i_4 = j_1 + j_2 + j_3 + j_4$, then we also have $i_4 = j_4$. Hence $\mathbf{u}$ and $\mathbf{v}$ are the same quintet, and then another contradiction occurs. Thus, the lemma is proved. $\square$

A construction of $\mathcal{U}$ by a graph-theoretical approach. Next, we show by Brooks' Theorem [35] that the size of the desired set of quintets $\mathcal{U}$ is at least $0.04\binom{n}{4}$, which improves the lower bound on the size of $\mathcal{U}$ in the previous simple construction.

Lemma 3.2. *There exists a set of quintets $\mathcal{U}$ over S which has size of at least $0.04\binom{n}{4}$ such that every two of them do not share a quartet.*

Proof. Let $G(V, E)$ be a graph such that vertices in V represent all quintets over the taxon set S, where two vertices u, v are adjacent if their corresponding quintets share a quartet. Then the degree of each vertex of G is bounded by $5(n-5)$. By Brooks' Theorem [35], the chromatic number of G is at most $5(n-5)$. Therefore by giving a proper coloring for G, we can derive that at least one color class (i.e., a set of monochromatic vertices of G) has size at least

$$\frac{\binom{n}{5}}{5(n-5)} = \frac{n(n-1)(n-2)(n-3)(n-4)}{5!(5(n-5))} > \frac{n(n-1)(n-2)(n-3)}{25 \cdot 4!} = 0.04\binom{n}{4}.$$

As each color class is an independent set, their corresponding quintets pairwise do not share a quartet. The lemma is proved. □

Theorem 3.1. *There exists a set of quartet topologies Q which is at least* 0.04*-far from being tree-like.*

Proof. Let Q^* be a tree-like set of quartet topologies over the taxon set S. We know that there exists a set of quintets $\mathcal{U}$ over S of size at least $0.04\binom{n}{4}$ such that every two quintets in $\mathcal{U}$ do not share any quartet (by Lemma 3.2). Then, for each quintet in $\mathcal{U}$, we arbitrarily pick one of its subset quartets and change its corresponding topology in Q^* to one of the other two possible topologies arbitrarily. Let Q denote the resulting set of quartet topologies. Now, every quintet in $\mathcal{U}$ with respect to Q has exactly one subset quartet whose topology is changed. Since one has to change at least two quartet topologies over a resolved quintet to make this quintet resolved again (by Fact 3.1), every quintet in $\mathcal{U}$ is unresolved with respect to Q. Furthermore, for each of these unresolved quintets in $\mathcal{U}$, we have to change at least one quartet topology of its subset quartets to make it resolved (otherwise, the unresolved quintet stays the same). Hence at least $|\mathcal{U}|$ quartet topologies in Q have to be changed to make the unresolved quintets in $\mathcal{U}$ with respect to Q resolved. Therefore, we obtain that the error number of Q is at least $|\mathcal{U}| \geq 0.04\binom{n}{4}$, hence Q is at least 0.04-far from being tree-like, as claimed by the theorem. □

3.3 An $O(n^3/\epsilon)$ Property Tester

Our property tester for tree-likeness of quartet topologies, denoted by Tree-Like-Tester, is presented in Algorithm 3.1. Theorem 2.4 is used as the building block of our property tester.

Tree-Like-Tester(Q) /* Q: a complete set of quartet topologies. */
begin
pick an arbitrary taxon $\ell \in S$;
repeat
  pick four taxa $s_1, s_2, s_3, s_4 \in S \setminus \{\ell\}$ uniformly at random;
  if the quintet $\{s_1, s_2, s_3, s_4, \ell\}$ is not resolved **then**
    return "no";
  end if
until the loop iterates for $\frac{72}{\epsilon}n^3$ times
return "yes".
end

Algorithm 3.1: Tree-Like-Tester: a property tester for testing tree-likeness of quartet topologies.

Remarks. It follows from Theorem 2.4 that we can determine whether Q is tree-like by examining quintets with respect to Q. If Q is not tree-like (i.e., the error number of Q is at least one), by Theorem 2.4, we know that for any fixed taxon $\ell \in S$, there exists an unresolved quintet containing ℓ. Hence it is clear that the number of unresolved quintets with respect to Q is at least $\Omega(n)$, which yields an $O(n^4)$ deterministic algorithm to see if Q is tree-like. Intuitively, we expect more unresolved quintets when the error number of Q gets larger. In particular, if the error number of Q is at least cn^4 for some constant c, we expect to have a large number (e.g., $c'n^5$ for some constant c') of unresolved quintets with respect to Q since each quartet is contained in $n-4$ quintets. The more unresolved quintets exist, the less queries are required to find one of them. However, it is difficult to give an accurate estimate of the number of unresolved quintets due to the following reason. Assume that Q^* is a tree-like set of quartet topologies such that $|Q \setminus Q^*|$ is equal to the error number of Q. Clearly, Q can be derived from Q^* by changing the quartet topologies in $Q^* \setminus Q$ one by one. However, changing a quartet topology may either make a set of unresolved quintets resolved or make a set of resolved quintets unresolved. After $|Q \setminus Q^*|$ changes, it is difficult to say how many unresolved quintets exist with respect to Q.

We now consider the case that Q is ϵ-far from being tree-like. That is, one has to change at least $\epsilon\binom{n}{4}$ quartet topologies to make Q tree-like. The following theorem provides an improved lower bound on the number of unresolved quintets.

Theorem 3.2. *If Q is ϵ-far from being tree-like, then for an arbitrary taxon $\ell \in S$, there exist more than $\epsilon n/36$ unresolved quintets containing ℓ.*

Proof. Assume that Q is ϵ-far from being tree-like. First, fix an arbitrary taxon ℓ. Let S^* be a maximal subset of S containing ℓ such that the subset Q_{S^*} of Q over S^* is tree-like, and let $S' = S \setminus S^*$. It is clear that adding any further taxon of S' into S^* will cause inconsistency (i.e., the set of quartet topologies over S^* is not tree-like). The size of S' can never be $o(n)$, otherwise, Q can be modified to be tree-like by simply changing the quartet topologies $\{[a_1a_2|a_3b] \mid a_1, a_2, a_3 \in S^*, b \in S'\} \cup \{[a_1a_2|b_1b_2] \mid a_1, a_2 \in S^*, b_1, b_2 \in S'\} \cup \{[ab_1|b_2b_3] \mid a \in S^*, b_1, b_2, b_3 \in S'\} \cup \{[b_1b_2|b_3b_4] \mid b_1, b_2, b_3, b_4 \in S'\}$, and the number of these changes of quartet topologies is at most

$$\binom{n-o(n)}{3}\cdot\binom{o(n)}{1}+\binom{n-o(n)}{2}\cdot\binom{o(n)}{2}+\binom{n-o(n)}{1}\cdot\binom{o(n)}{3}+\binom{o(n)}{4}=o(n^4),$$

which contradicts the assumption that the error number of Q is at least $\epsilon\binom{n}{4}$. Thus we let the size of S' be αn, where $\alpha > 0$ is a constant. $(S^* \cup \{x\})$ must have at least one unresolved quintet containing ℓ for every taxon $x \in S'$ (by Theorem 2.4 and the assumption that S^* is maximal). Therefore, the number of unresolved quintets containing an arbitrarily fixed taxon ℓ with respect to Q is at least αn. By the previous discussion, we know that the number of quartet topologies we need to change to make Q tree-like is at most

$$\begin{aligned} & \sum_{i=0}^{4}\binom{n-\alpha n}{4-i}\binom{\alpha n}{i}-\binom{n-\alpha n}{4} \\ = & \binom{n}{4}-\binom{n-\alpha n}{4} \quad \text{(by Vandermonde's identity).} \end{aligned}$$

Since the error number of Q is at least $\epsilon\binom{n}{4}$, we have

$$\epsilon\binom{n}{4} \le \binom{n}{4}-\binom{n-\alpha n}{4} \le \frac{n^4}{24}-\left(\frac{(1-\alpha)n}{2}\right)^4 < \frac{n^4(1-(1-\alpha)^4)}{16}.$$

So we obtain that

$$\epsilon < \frac{n^4(1-(1-\alpha)^4)/16}{n(n-1)(n-2)(n-3)/24} = \frac{n^4(3/2)(1-(1-\alpha)^4)}{n(n-1)(n-2)(n-3)} \le 9(1-(1-\alpha)^4).$$

The last inequality holds since $n \geq 5$ and $n(n-1)(n-2)(n-3) \geq n^4/6$ for $n \geq 5$. Hence by using Taylor series expansion, we have $\alpha > 1 - (1-\epsilon/9)^{1/4} = 1 - (1 - \epsilon/36 - \epsilon^2/864 - \ldots) > \epsilon/36$. Therefore the theorem follows. □

The following theorem shows that Algorithm Tree-Like-Tester, is a non-adaptive property tester which is of one-sided error and makes at most $O(n^3/\epsilon)$ queries.

Theorem 3.3. *Algorithm* Tree-Like-Tester *is a non-adaptive and one-sided-error property tester for tree-likeness of quartet topologies, which makes at most $O(n^3/\epsilon)$ queries.*

Proof. If Q is tree-like, then the algorithm will never find an unresolved quintet, it will always return "yes", hence it is of one-sided-error. As for the case that Q is ϵ-far from being tree-like, consider an arbitrarily fixed taxon $\ell \in S$. By Theorem 3.2, the number of unresolved quintets containing ℓ with respect to Q is more than $\epsilon n/36$. In each iteration of the loop of the algorithm, the probability of finding an unresolved quintet is at least

$$\frac{\epsilon n/36}{\binom{n-1}{4}} \geq \frac{(\epsilon/36)}{n^3}.$$

For simplicity, let α denote $(\epsilon/36)/n^3$. Once an unresolved quintet is found during these $2/\alpha$ iterations, the algorithm returns "no", otherwise, it returns "yes", with probability at most $(1-\alpha)^{2/\alpha} \leq e^{-2} < 1/3$, where we use the fact that $(1-t)^{-1/t} \geq e$ for any $t > 0$ (Note that $e^{-1} = \lim_{t\to 0}(1-t)^{1/t}$). Moreover, checking whether a quintet is resolved or not requires at most five queries, thus at most $O(n^3/\epsilon)$ queries are made by the algorithm. Since the algorithm makes each query without knowing the results of previous ones, it is clearly non-adaptive. The theorem is proved. □

3.4 The Difficulty of Testing Tree-Consistency by Examining Quintets

We propose a one-sided error property tester for tree-likeness of quartet topologies, which is non-adaptive and utilizes at most $O(n^3/\epsilon)$ queries. However, for the moment, whether the query complexity of testing tree-likeness of quartet topologies can be proved to be independent of n still remains open.

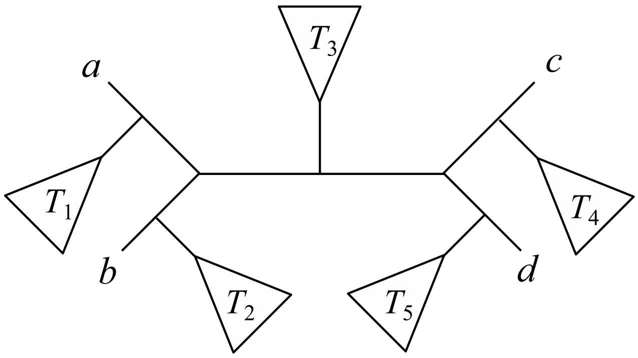

Figure 3.1: The tree structure with the quartet topology $[ab|cd]$. T_1, T_2, T_3, T_4, and T_5 are subtrees.

One might be curious about whether our results can be extended to incomplete sets of quartet topologies. Unfortunately, it seems to be impossible since Theorem 2.4 is not true when the set of quartet topologies Q is incomplete. Let us say a quintet is *partially resolved* if the set of quartet topologies over this quintet in Q is tree-consistent (but not necessarily tree-like). The following example illustrates that there exists an incomplete set of quartet topologies Q, such that Q is not tree-consistent even when each quintet is partially resolved with respect to Q.

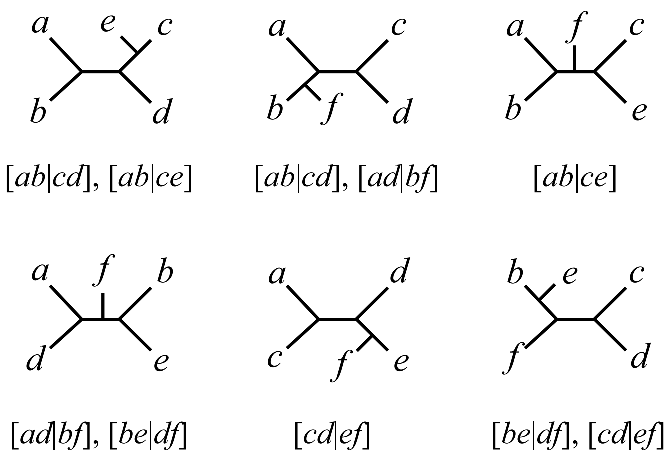

Figure 3.2: $Q = \{[ab|cd], [ab|ce], [ad|bf], [be|df], [cd|ef]\}$. Each quintet over $S = \{a, b, c, d, e, f\}$ is partially resolved.

Let $Q = \{[ab|cd], [ab|ce], [ad|bf], [be|df], [cd|ef]\}$ be a set of quartet topologies over $S = \{a, b, c, d, e, f\}$. Obviously, Q is not complete. The $\binom{6}{5} = 6$ quintets over S are $\{a, b, c, d, e\}$, $\{a, b, c, d, f\}$, $\{a, b, c, e, f\}$, $\{a, b, d, e, f\}$, $\{a, c, d, e, f\}$, and $\{b, c, d, e, f\}$. Let us first observe the possible topologies of the quintet $\{a, b, c, d, e\}$.

Fig. 3.1 depicts the evolutionary tree with the quartet topology $[ab|cd]$. Since $[ab|ce] \in Q$, as Fig. 3.1 shows, e has to be in T_3, T_4, or T_5. Similarly, f has to be in T_2. Then the induced topology of the quartet $\{b, d, e, f\}$ on the evolutionary tree can only be $[bf|de]$. Since this conflicts with the assumption that $[be|df] \in Q$, we derive that Q is not tree-consistent (Q is clearly not tree-like since Q is incomplete). However, as Fig. 3.2 shows, each of these six quintets is partially resolved.

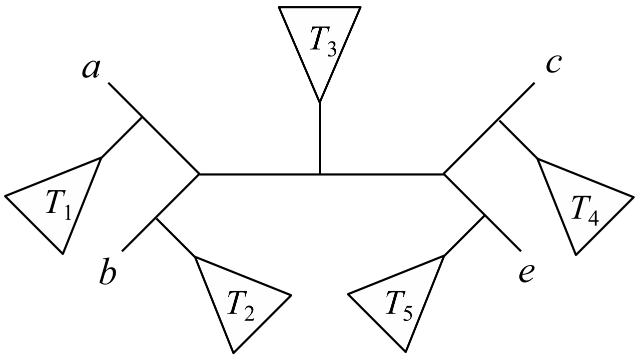

Figure 3.3: The tree structure with the quartet topology $[ab|ce]$.

In the above example, each quintet has at most two of its subset quartets with topologies in Q. One might conjecture that if the input Q is "dense enough", that is, almost all the subset quartets of each quintet have topologies in Q, then we might be able to derive that Q is tree-consistent if and only if each quintet is partially resolved. However, the following example disproves this conjecture. Let $Q = \{[ab|ce], [ac|bf], [ab|de], [ad|bf], [ae|bf], [ad|ce], [ac|df], [af|ce], [bd|ce], [bf|cd], [bf|ce], [bf|de], [ce|df]\}$ be a set of quartet topologies over $S = \{a, b, c, d, e, f\}$. There are only two quartets which do not have topologies in Q (i.e., $\{a, b, c, d\}$ and $\{a, d, e, f\}$). We observe that Q is "dense" in this case. To be precise, for each quintet, at least four of its subset quartets have topologies in Q. Similar to the previous example, we observe from Fig. 3.4 that each quintet is partially resolved. However, Q is not tree-consistent due to the following observation. Consider the topology of the quintet $\{a, b, c, d, e\}$. The evolutionary tree with the quartet topology $[ab|ce]$ is depicted in Fig. 3.3. Since $[ab|de], [ad|ce] \in Q$, the taxon d has to be in T_3. Similarly, we derive that f has to be in T_2 since $[ae|bf] \in Q$. Then we obtain that the induced topology of the quartet $\{a, c, d, f\}$ on the evolutionary tree can only be $[af|cd]$, which contradicts the assumption that $[ac|df] \in Q$.

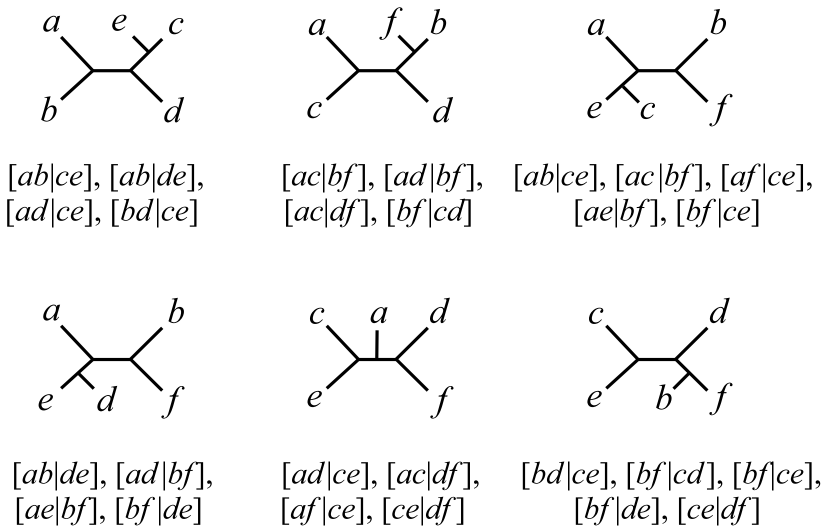

Figure 3.4: A set of thirteen quartet topologies Q, where only two quartets $\{a, b, c, d\}$ and $\{a, d, e, f\}$ do not have topologies in Q. Each quintet over $S = \{a, b, c, d, e, f\}$ is partially resolved.

By the above two examples, we conclude that when the input set of quartet topologies is not complete, "local consistency" (i.e., the property that each quintet is partially resolved) does not guarantee "global consistency" (i.e., the property that Q is tree-consistent).

Chapter 4

Testing Tree-Consistency with at Most k Missing Quartets

In the end of Chapter 3, we learned that the naïve approach of sampling of quintets over an n-taxon set S and then examining if they are resolved with respect to the set Q of quartet topologies does not work for testing tree-consistency when Q is incomplete, even Q is quite dense. In this chapter, we extend the previous result by introducing a parameter k into the testing for tree-consistency, where k denotes an upper bound on the number of the quartets whose topologies are missing with respect to Q. We present two parameterized property testers for tree-consistency with respect to such a parameter. Both of them are non-adaptive, have one-sided error, and are uniform on k. The first one runs in $O(3^k kn^3/\epsilon)$ time, and the second one runs in $O(1.7321^k kn^3/\epsilon)$ time.

By the parameterized property testing results, we obtain that the number of quartets whose topologies are missing is a factor which makes the testing difficult. To some degree, this coincides with the fact that determining if a set of quartet topologies is tree-consistent is **NP**-complete when some quartet topologies over S are missing [110]. The results in this chapter also complete our assertion that the problem of determining consistency of quartet topologies can be efficiently solved through the aspects of fixed-parameter algorithm, property testing, and parameterized property testing.

The setting of property testing for tree-consistency. We introduce the setting of property testing for tree-consistency which is similar to the one for tree-likeness in Sect. 3.1. Let $\prec$ be a total order on the n-taxon set S, and Q be a set of quartet topologies over S. For the three possible topologies of a quartet $\{a,b,c,d\}$, we denote the quartet topologies $[ab|cd]$, $[ac|bd]$, and $[ad|bc]$ by type 0, 1, and 2, respectively, where $a \prec b \prec c \prec d$ (as defined in Sect. 2.3.1). A quartet over S is called a *missing quartet* if it does not have a topology in Q. We use a function $\hat{f}_Q : \{\{a,b,c,d\} \mid a,b,c,d \in S\} \mapsto \{0,1,2,\varnothing\}$ to represent Q as well as the missing quartets. The function value $\hat{f}_Q(\{a,b,c,d\}) \neq \varnothing$ is equal to the type of the topology of $\{a,b,c,d\}$ in Q, and $\hat{f}_Q(\{a,b,c,d\}) = \varnothing$ denotes that the quartet $\{a,b,c,d\}$ is a missing quartet. The domain size of the function $\hat{f}_Q$ is then equal to $\binom{n}{4}$. Note that $\hat{f}_Q$ is exactly the function f_Q defined in Chapter 3 when there are no missing quartets. We regard $\hat{f}_Q$ and an integer k as the input, where k is the number of missing quartets. Each query of $\hat{f}_Q$ retrieves the topology of a quartet in Q or simply the null symbol $\varnothing$. We utilize an array of $\binom{n}{4}$ entries, where each entry stores a function value of $\hat{f}_Q$. Only the function values of $\hat{f}_Q$ which are not $\varnothing$ are allowed to be modified. Changing a topology in Q of a quartet to another one corresponds to modifying a function value of $\hat{f}_Q$. Recall that $\mathcal{P}_{tree}$ denotes the set of all the functions f_Q for tree-like Q's. We define that $\Delta(Q,\mathcal{P}_{tree}) = \min_{Q^* \in \mathcal{P}_{tree}} |Q \setminus Q^*|/\binom{n}{4}$. We say that $\hat{f}_Q$ is tree-consistent if $\Delta(\hat{f}_Q,\mathcal{P}_{tree}) = 0$ (i.e., Q is tree-consistent), and $\hat{f}_Q$ is ϵ-far from being tree-consistent if $\Delta(\hat{f}_Q,\mathcal{P}_{tree}) \geq \epsilon$ (i.e., the error number of Q is at least $\epsilon\binom{n}{4}$). Testing if Q is tree-consistent turns to testing if $\hat{f}_Q$ is tree-consistent. When the context clear, we simply say that Q is tree-consistent (resp., Q is ϵ-far from being tree-consistent) if $\hat{f}_Q$ is tree-consistent (resp., $\hat{f}_Q$ is ϵ-far from being tree-consistent). Note that if $|Q| < \epsilon\binom{n}{4}$, then Q is ϵ-close to being tree-consistent since one can modify less than $\epsilon\binom{n}{4}$ quartet topologies in Q to make Q tree-consistent. For such a case, since Q can never be ϵ-far from being tree-consistent, testing if Q is tree-consistent becomes trivial since one can always answer "yes". Due to this reason, we assume that the size of Q is at least $\epsilon\binom{n}{4}$.

The rest of this chapter is organized as follows. In Sect. 4.1, we introduce an $O(3^k kn^3/\epsilon)$ parameterized property tester, which is called TC-Tester, for tree-consistency of quartet topologies with at most k missing quartets. The tester has one-sided error and non-adaptive. Based on this tester, in Sect. 4.2 we give an improved parameterized property tester, which is called Improved-TC-Tester, for this property. The tester has time complexity of $O(1.7321^k kn^3/\epsilon)$.

4.1 An $O(3^k kn^3/\epsilon)$ Parameterized Property Tester

In this section, we propose an algorithm, which is denoted by TC-Tester, for testing tree-consistency of a set of quartet topologies according which there are at most k missing quartets. The sketch of the algorithm is as follows. There are two stages of the algorithm: the *sampling stage* and the *testing stage*. In the sampling stage, the algorithm chooses a taxon ℓ arbitrarily from S, and then samples a multiset of quintets over S which contain ℓ uniformly at random. These quintets are collected into two sets $\mathcal{F}_1$ and $\mathcal{F}_2$, where the selected quintets which do not contain any missing quartet are in $\mathcal{F}_1$ and the other ones which contain missing quartets are in $\mathcal{F}_2$. Then, the algorithm enters the testing stage. If any quintet in $\mathcal{F}_1$ is unresolved, then it returns "no", otherwise it continues to examine the quintets in $\mathcal{F}_2$. There are at most k missing quartets found in the sampling stage. Since a quartet has three possible topologies, there are at most 3^k possible assignments of the topologies of the found missing quartets. We call them *topology assignments* for short. The algorithm exhaustively tries all of these possible assignments, which are generated and stored in a set $\mathcal{A}$, and returns "yes" if there is one of them under which all the quintets in $\mathcal{F}_2$ are resolved. It returns "no" if there is no such assignment. The pseudocode of the algorithm is listed in Algorithm 4.1.

For the analysis of Algorithm TC-Tester, we utilize Theorem 3.2, which provides a lower bound on the number of unresolved quintets containing a fixed taxon with respect to a complete set of quartet topologies which is ϵ-far from being tree-like.

Theorem 4.1. *Given a set Q of quartet topologies over an n-taxon set S where there are at most k missing quartets, Algorithm* TC-Tester *is an $O(3^k kn^3/\epsilon)$ parameterized property tester with one-sided error for testing if Q is tree-consistent. Moreover, it has one-sided error, is non-adaptive and is uniform on k.*

Proof. Sampling quintets (Lines 3–13) takes $O(kn^3/\epsilon)$ time. To determine if a quintet without having any missing quartet can be done in $O(1)$ time, so examining if any quintet in $\mathcal{F}_1$ is unresolved takes $O(|\mathcal{F}_1|)$ time. When the missing quartets in the sampled quintets are obtained, to generate all possible assignments of their topologies (at Line 23) requires $O(3^{|\mathcal{T}_{miss}|}) = O(3^k)$ time. To check if all the quintets in $\mathcal{F}_2$ are resolved for any of the $O(3^k)$ topology assignments of the found missing

TC-Tester(Q, k) /* Q: a set of quartet topologies;
$k \in \mathbb{Z}^+$: an upper bound on the number of missing quartets. */
begin
/* *Sampling Stage* */
pick an arbitrary taxon $\ell \in S$;
repeat
    pick a quartet $\{s_1, s_2, s_3, s_4\}$ over $S \setminus \{\ell\}$ uniformly at random;
    let $\boldsymbol{u}$ denote the quintet $\{s_1, s_2, s_3, s_4, \ell\}$;
    if $\boldsymbol{u}$ does not contain any missing quartet **then**
        $\mathcal{F}_1 \leftarrow \mathcal{F}_1 \cup \{\boldsymbol{u}\}$; /* $\mathcal{F}_1 \leftarrow \emptyset$ initially */
    else /* $\boldsymbol{u}$ contains a missing quartet */
        $\mathcal{F}_2 \leftarrow \mathcal{F}_2 \cup \{\boldsymbol{u}\}$; /* $\mathcal{F}_2 \leftarrow \emptyset$ initially */
        $\text{miss}(\boldsymbol{u}) \leftarrow \{\text{missing quartets of } \boldsymbol{u}\}$;
        $\mathcal{T}_{miss} \leftarrow \mathcal{T}_{miss} \cup \text{miss}(\boldsymbol{u})$; /* $\mathcal{T}_{miss} \leftarrow \emptyset$ initially; it collects missing quartets */
    end if
until the loop iterates for $144(k+1)n^3/\epsilon$ times
/* *Testing Stage* */
for each quintet $\boldsymbol{u} \in \mathcal{F}_1$ **do**
    if $\boldsymbol{u}$ is NOT resolved **then**
        return "no";
    end if
end for
if $\mathcal{F}_2 = \emptyset$ **then** /* no missing quartet is found */
    return "yes";
else
    generate the set of all possible topology assignments $\mathcal{A} = \{Q_{miss}(i) \mid 1 \leq i \leq 3^{|\mathcal{T}_{miss}|}\}$ of the missing quartets in $\mathcal{T}_{miss}$;
    for each assignment $Q_{miss}(i)$ **do**
        if ALL the quintets in $\mathcal{F}_2$ are resolved with respect to $Q \cup Q_{miss}(i)$ **then**
            return "yes";
        end if
    end for
    return "no";
end if
end

Algorithm 4.1: TC-Tester: a parameterized property tester for tree-consistency quartets (the for-loop in Lines 24–28) takes $O(3^k \cdot |\mathcal{F}_2|)$ time. Since $|\mathcal{F}_1| + |\mathcal{F}_2| = O(kn^3/\epsilon)$, the overall time complexity of Algorithm TC-Tester is $O(|\mathcal{F}_1| + |\mathcal{F}_2| + 3^k \cdot |\mathcal{F}_2|) = O(3^k kn^3/\epsilon)$.

Next, we prove the correctness of Algorithm TC-Tester as follows. First, consider the case that Q is tree-consistent. By definition, there exists an evolutionary tree T such that $Q \subset Q_T$. Since the algorithm exhaustively tries every assignment of topologies for the missing quartets in $\mathcal{T}_{miss}$ (Line 23), there must be some $i \in \{1, 2, \ldots, 3^{|\mathcal{T}_{miss}|}\}$ such that $Q_{miss}(i) \subseteq Q_T \setminus Q$ for the ith topology assignment $Q_{miss}(i)$. Thus the algorithm must return "yes" in this case (hence it has one-sided error). Consider the case that Q is ϵ-far from being tree-consistent. Let $\mathcal{T}^*_{miss}$ be the set of missing quartets with respect to Q. For any assignment of the topologies, say Q^*_{miss}, of the missing quartets in $\mathcal{T}^*_{miss}$, it is clear that $Q \cup Q^*_{miss}$ becomes a complete set of quartet topologies over S and has at least $\epsilon\binom{n}{4}$ quartet errors, hence $Q \cup Q^*_{miss}$ is ϵ-far from being tree-like. Note that $\mathcal{T}_{miss} \subseteq \mathcal{T}^*_{miss}$, and $Q_{miss}(i) \subseteq Q^*_{miss}$ for some $i \in (1, 2, \ldots, 3^{|\mathcal{T}_{miss}|})$. By Theorem 3.2, the probability that a randomly sampled quintet containing a fixed taxon ℓ is unresolved with respect to $Q \cup Q^*_{miss}$ is at least $(\epsilon n/36)/(\binom{n-1}{4}) > \epsilon n^{-3}/36$. Denote $\epsilon n^{-3}/36$ by α. The algorithm returns "yes" in this case only when the following two events both occur:

(C1) all the quintets in $\mathcal{F}_1$ are resolved (Lines 15–19);

(C2) there exists a topology assignment of the found missing quartets such that all the quintets in $\mathcal{F}_2$ are resolved (Lines 20–30).

The event of (C1) $\cap$ (C2) is equivalent to the following event.

(C3) there exists an topology assignment of the found missing quartets such that all the quintets in $\mathcal{F}_1 \cup \mathcal{F}_2$ are resolved.

Since each quintet in $\mathcal{F}_1 \cup \mathcal{F}_2$ is sampled independently, for each iteration of the loop in Lines 24–28, all the quintets in $\mathcal{F}_1 \cup \mathcal{F}_2$ are resolved with probability at most

$(1-\alpha)^{|\mathcal{F}_1 \cup \mathcal{F}_2|}$. Thus by the union bound, the probability of (C3) is at most

$$\begin{aligned}(1-\alpha)^{|\mathcal{F}_1 \cup \mathcal{F}_2|} \cdot 3^{|\mathcal{T}_{miss}|} &\leq (1-\alpha)^{2(k+1)/\alpha} \cdot 3^k \\ &< (e^{-2})^{k+1} \cdot 3^k \\ &< \left(\frac{1}{3}\right)^{k+1} \cdot 3^k \\ &< \frac{1}{3}.\end{aligned}$$

Therefore, for the case that Q is ϵ-far from being tree-consistent, the algorithm returns "yes" with probability less than 1/3.

Since each quintet is sampled without knowing the previous ones (see Lines 3–13 of the algorithm), the algorithm is non-adaptive. Furthermore, it uses a unified approach for all k's, hence it is uniform on k. Thus, the theorem is proved.

□

4.2 An $O(1.7321^k kn^3/\epsilon)$ Parameterized Property Tester

At Line 23 of Algorithm TC-Tester, all the possible $3^{|\mathcal{T}_{miss}|}$ topology assignments of the missing quartets in $\mathcal{T}_{miss}$ are generated in order to check if all the quintets in $\mathcal{F}_2$ are resolved with respect to some topology assignment. This guarantees that the algorithm always answers "yes" when Q is tree-consistent. Consider a quintet with $\binom{5}{4} = 5$ missing quartets. There are $3^5 = 243$ possible assignments of the topologies of these five quartets. However, as Fig. 2.3 shows explicitly, there are only fifteen of them which make the quintet resolved. Such an observation suggests that it may not need to exhaustively try all the $3^{|\mathcal{T}_{miss}|}$ topology assignments of the missing quartets, where $3^{|\mathcal{T}_{miss}|}$ may be up to 3^k. Based on this idea, we improve that complexity of Algorithm TC-Tester by generating a smaller set of topology assignments of the found missing quartets, which is of size bounded by 1.7321^k.

Recall that there are fifteen possible quintet topologies for a quintet $\{s_1, s_2, s_3, s_4, s_5\}$ (see Fig. 2.3) and $\mathcal{V}$ denotes the set of topology vectors of all the possible quintet topologies of a quintet. In particular,

$$\mathcal{V} = \left\{ \begin{array}{lllll} (0,2,2,2,2), & (0,1,1,2,2), & (0,0,0,2,2), & (0,0,0,1,1), & (0,0,0,0,0), \\ (1,2,2,2,1), & (1,0,2,1,1), & (1,1,2,0,1), & (1,1,1,0,2), & (1,1,0,0,0), \\ (2,2,2,2,0), & (2,2,0,0,0), & (2,2,1,1,0), & (2,1,1,1,2), & (2,0,1,1,1). \end{array} \right\},$$

which corresponds to the quintet topologies in Fig. 2.3 by letting $s_1 = a$, $s_2 = b$, $s_3 = c$, $s_4 = d$, and $s_5 = e$ respectively.

Lemma 4.1. *Let $\boldsymbol{u} \subseteq S$ be a quintet with r missing quartets $\{\boldsymbol{q}_i \mid 1 \leq i \leq 5\}$ with respect to a set Q of quartet topologies over S, then there exists at most β_r topology assignments of these missing quartets which can make $\boldsymbol{u}$ resolved, where $\beta_1 = 1$, $\beta_2 = 3$, $\beta_3 = 3$, $\beta_4 = 5$, and $\beta_5 = 15$.*

Proof. Without loss of generality, let $\boldsymbol{u} = \{a, b, c, d, e\}$. First, we consider the case that $r = 1$, that is, $\boldsymbol{u}$ contains one missing quartet. Assume that the missing quartet of $\boldsymbol{u}$ is $\{a, b, c, d\}$. Let $(v_{abce}, v_{abde}, v_{acde}, v_{bcde})$ be a vector, where v_{abce} denotes the type of the topology of $\{a, b, c, e\}$, v_{abde} denotes the type of the topology of $\{a, b, d, e\}$, v_{acde} denotes the type of the topology of $\{a, c, d, e\}$, and v_{bcde} denotes the type of the topology of $\{b, c, d, e\}$. For $\boldsymbol{u}$ to be resolved, from the list $\mathcal{V}$ we know that there are fifteen possibilities of $(v_{abce}, v_{abde}, v_{acde}, v_{bcde})$. For each possibility of $(v_{abce}, v_{abde}, v_{acde}, v_{bcde})$, there is exactly one possible assignment of the topology of the missing quartet $\{a, b, c, d\}$ to make $\boldsymbol{u}$ resolved. For example, assume that $(v_{abce}, v_{abde}, v_{acde}, v_{bcde}) = (2, 2, 2, 2)$, then from $\mathcal{V}$ we obtain that $\{a, b, c, d\}$ must have the topology $[ab|cd]$ (i.e., the topology of type 0), otherwise $\boldsymbol{u}$ cannot be resolved. Similarly, for the cases that the missing quartet is either $\{a, b, c, e\}$, $\{a, b, d, e\}$, $\{a, c, d, e\}$ or $\{b, c, d, e\}$, we obtain that there is at most one assignment of its topology to make $\boldsymbol{u}$ resolved. Hence, we have $\beta_1 = 1$.

Consider the case that $r = 2$. Assume that the missing quartets of $\boldsymbol{u}$ are $\{a, b, c, d\}$ and $\{a, b, c, e\}$. Similar to the previous paragraph, we let $(v_{abde}, v_{acde}, v_{bcde})$ denote a vector, where v_{abde} denotes the type of the topology of $\{a, b, d, e\}$, v_{acde} denotes the type of the topology of $\{a, c, d, e\}$, and v_{bcde} denotes the type of the topology of $\{b, c, d, e\}$. For $\boldsymbol{u}$ to be resolved, from the list $\mathcal{V}$ we know that there are thirteen possibilities of $(v_{abde}, v_{acde}, v_{bcde})$. For each possibility of $(v_{abde}, v_{acde}, v_{bcde})$, there are at most three possible assignments of the topologies of the missing quartets $\{\{a, b, c, d\}, \{a, b, c, e\}\}$ to make $\boldsymbol{u}$ resolved. For example, assume that $(v_{abde}, v_{acde}, v_{bcde}) = (0, 0, 0)$, then from $\mathcal{V}$ we obtain that the topology of $\{a, b, c, d\}$ and $\{a, b, c, e\}$ must be $[ab|cd]$ and $[ab|ce]$, or $[ac|bd]$ and $[ac|be]$, or $[ad|bc]$ and $[ae|bc]$, otherwise $\boldsymbol{u}$ cannot be resolved. For the other cases of two missing quartets, similar results can be derived. Hence, we obtain that $\beta_2 = 3$.

Similar to the arguments in the previous paragraph, we obtain that $\beta_3 = 3$ and $\beta_4 = 5$. For the case that $r = 5$, since all the quartets in $\boldsymbol{u}$ are missing, we have $\beta_5 = 15$, which is exactly the number of possible topologies of a quintet. Therefore, the lemma is proved.

□

Instead of using the naïve approach of exhaustively trying all the $3^{|\mathcal{T}_{miss}|}$ topology assignments, in the following we consider another approach, which is based on Lemma 4.1, to generate the set of possible topology assignments of $\mathcal{T}_{miss}$ which contain all the assignments under each of which all the quintets in $\mathcal{F}_2$ are resolved. We call such a set, denoted by $\mathcal{A}^{LR}$, the *least required* set of topology assignments.

The approach for generating $\mathcal{A}^{LR}$ works as a recursive algorithm and can be regarded as a depth-bounded search tree. The number of recursion calls is the number of nodes in the search tree. First, for each found missing quartet $\boldsymbol{q}$, we collect the picked quintets which contain $\boldsymbol{q}$ into a set $\mathcal{L}(\boldsymbol{q})$. Then, get a copy $\mathcal{F}'$ of $\mathcal{F}_2$. For each quintet $\boldsymbol{u} \in \mathcal{F}'$, we recursively branch on the possible topology assignments of its missing quartets according to the list $\mathcal{V}$ in order to make $\boldsymbol{u}$ resolved. Denote by $\text{miss}(\boldsymbol{u})$ the set of missing quartets in $\boldsymbol{u}$. In each branch, the topologies of the missing quartets in $\text{miss}(\boldsymbol{u})$ are determined, and a quintet in $\mathcal{L}(\boldsymbol{q})$, for $\boldsymbol{q} \in \text{miss}(\boldsymbol{u})$, is removed from $\mathcal{F}'$ if all its missing quartets are assigned with topologies. The number of such quintet removals is at most $O(n)$ due to the reason that there are $O(n)$ quintets that contain a fixed quartet. The algorithm stops branching if either all the missing quartets have topologies determined, or the current examined quintet $\boldsymbol{u} \in \mathcal{F}'$ can never be resolved no matter what topology assignment of the missing quartets of $\boldsymbol{u}$ is. For the former case, we add the according topology assignment of $\mathcal{T}_{miss}$ into $\mathcal{A}^{LR}$.

Let $N(k)$ denote the number of leaf nodes of the search tree. The size of the set $\mathcal{A}^{LR}$ is bounded by $N(k)$. By Lemma 4.1, we obtain the following recursive formula:

$$\begin{cases} N(k) \leq \max\{N(k-1), 3N(k-2), 3N(k-3), 5N(k-4), 15N(k-5)\}, \\ N(0) = 1. \end{cases}$$

For example, the inequality $N(k) \leq 3N(k-2)$ stands for the case that the examined quintet $\boldsymbol{u}$ has two missing quartets. By Lemma 4.1, there are at most three topology assignments of these two quartets, so the tree node according to examining $\boldsymbol{u}$ has at most three branches each of which has two less quartets with topologies not assigned yet. Let γ denote the branching number of the search tree. For $N(k) \leq N(k-1)$, the search tree does not branch. For $N(k) \leq 3N(k-2)$, we obtain a branching vector of $(2, 2, 2)$ which leads to $\gamma < 1.7321$. For $N(k) \leq 3N(k-3)$, we obtain a branching vector of $(3, 3, 3)$ which leads to $\gamma < 1.4423$. For $N(k) \leq 5N(k-4)$, we obtain a branching vector of $(4, 4, 4, 4, 4)$ which leads to $\gamma < 1.4954$. Finally, for $N(k) \leq 15N(k-5)$ we obtain a branching vector of $(5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5)$, which leads to $\gamma < 1.7188$. Therefore, the size of $\mathcal{A}^{LR}$ is bounded by 1.7321^k and the size of the according search tree is bounded by $O(1.7321^k)$. Since $|\mathcal{F}'| = O(kn^3/\epsilon)$, and, for each tree node, the algorithm takes $O(n)$ time to remove the quintets in $\mathcal{F}'$ without missing quartets, the overall time complexity of constructing $\mathcal{A}^{LR}$ is $O(1.7321^k n + kn^3/\epsilon)$.

By constructing the least required set $\mathcal{A}^{LR}$ of topology assignments of $\mathcal{T}_{miss}$, we obtain an improved property tester for testing tree-consistency of quartet topologies. The property tester is called Improved-TC-Tester and is listed in Algorithm 4.2.

As proved in the previous section, we know that it takes $O(1.7321^k n + kn^3/\epsilon)$ time to construct $\mathcal{A}^{LR}$ and the size of $\mathcal{A}^{LR}$ is bounded by 1.7321^k. Since Algorithm Improved-TC-Tester is basically the same as Algorithm TC-Tester, similar to the proof of Theorem 4.1, we obtain Theorem 4.2 as follows.

Theorem 4.2. *Given a set Q of quartet topologies over an n-taxon set S where there are at most k missing quartets, Algorithm* Improved-TC-Tester *is an $O(1.7321^k kn^3/\epsilon)$ property tester with one-sided for testing if Q is tree-consistent. Moreover, it has one-sided error, is non-adaptive and is uniform on k.*

Remarks. Our parameterized property testers run in $o(n^4)$ time when k is $o(\log n)$. By the results in this chapter, we obtain that tree-consistency of quartet topologies can be tested more efficiently when k gets smaller. This suggests that the number of missing quartets is a factor which makes the testing difficult. Actually, to determine if Q is tree-consistent (i.e., the QCP problem) is **NP**-complete [110] when missing quartets exist. However, the following arguments imply that it can be deterministically solved in polynomial time when the number of missing quartets is bounded by a constant k. First, we scan over the input to find out the missing quartet (it

Improved-TC-Tester(Q, k) /* Q: a set of quartet topologies;
$k \in \mathbb{Z}^+$: an upper bound on the number of missing quartets. */
begin
/* *Sampling Stage* */
pick an arbitrary taxon $\ell \in S$;
repeat
  pick a quartet $\{s_1, s_2, s_3, s_4\}$ over $S \setminus \{\ell\}$ uniformly at random;
  let $\boldsymbol{u}$ denote the quintet $\{s_1, s_2, s_3, s_4, \ell\}$;
  if $\boldsymbol{u}$ does not contain any missing quartet **then**
    $\mathcal{F}_1 \leftarrow \mathcal{F}_1 \cup \{\boldsymbol{u}\}$; /* $\mathcal{F}_1 \leftarrow \emptyset$ initially */
  else /* $\boldsymbol{u}$ contains a missing quartet */
    $\mathcal{F}_2 \leftarrow \mathcal{F}_2 \cup \{\boldsymbol{u}\}$; /* $\mathcal{F}_2 \leftarrow \emptyset$ initially */
    $\text{miss}(\boldsymbol{u}) \leftarrow \{$missing quartets of $\boldsymbol{u}\}$;
    for each missing quartet $\boldsymbol{q} \in \text{miss}(\boldsymbol{u})$ **do**
      $\mathcal{L}(\boldsymbol{q}) \leftarrow \mathcal{L}(\boldsymbol{q}) \cup \{\boldsymbol{u}\}$; /* $\mathcal{L}(\boldsymbol{q})$ collects the chosen quintets which contain the missing quartet $\boldsymbol{q}$; $\mathcal{L}(\boldsymbol{q}) \leftarrow \emptyset$ initially */
    end for
    $\mathcal{T}_{miss} \leftarrow \mathcal{T}_{miss} \cup \text{miss}(\boldsymbol{u})$; /* $\mathcal{T}_{miss} \leftarrow \emptyset$ initially; it collects missing quartets */
  end if
until the loop iterates for $144(k+1)n^3/\epsilon$ times
/* *Testing Stage* */
for each quintet $\boldsymbol{u} \in \mathcal{F}_1$ **do**
  if $\boldsymbol{u}$ is NOT resolved **then**
    return "no";
  end if
end for
if $\mathcal{F}_2 = \emptyset$ **then** /* no missing quartet is found */
  return "yes";
else
  generate the least required set of topology assignments $\mathcal{A}^{LR} = \{Q^{LR}_{miss}(i) \mid i \geq 1\}$ of the missing quartets in $\mathcal{T}_{miss}$;
  if $\mathcal{A}^{LR} \neq \emptyset$ **then**
    for each assignment $Q_{miss}(i)$ **do**
      if ALL the quintets in $\mathcal{F}_2$ are resolved with respect to $Q \cup Q_{miss}(i)$ **then**
        return "yes";
      end if
    end for
  end if
  return "no";
end if
end

Algorithm 4.2: Improved-TC-Tester: an improved parameterized property tester for tree-consistency takes $O(n^4)$ time here). Then, for each topology assignment $Q_{miss}(i)$ of the missing quartets, check if $Q \cup Q_{miss}(i)$ is tree-like. To check if $Q \cup Q_{miss}(i)$ is tree-like takes $O(n^4)$ since it is complete. Clearly, the above work takes $O(3^k n^4)$ time, and even $O(1.7321^k n^4)$ time when the least required set of topology assignments for the missing quartets is applied. This indicates that to determine if Q is tree-consistent is fixed-parameter tractable with respect to the parameter k, by which the number of missing quartets is bounded.

Chapter 5

Parameterized Property Testers for Graph Properties

In Chapter 4, we extend the property tester for tree-likeness to test tree-consistency on a set of quartet topologies with at most k missing quartets by parameterized property testing. This example illustrates how a parameterized property tester is designed, and reveals that the concepts of property testing and parameterized complexity theory can be fruitfully combined, so that we can tackle with hard problems making use of the advantages of these two fields.

As mentioned in Sect. 1.3 of Chapter 1, there have been several examples of graph property testing that fit our setting of parameterized property testing. In this chapter, we keep studying parameterized property testing for graph properties. Let us recall the settings of the dense model and the sparse model for graph property testing as follows.

The dense model. The dense model is suitable for dense graphs. In this model, *adjacency-matrices* are commonly used as the representation of graphs. A property tester is allowed to make queries, where each query is to examine the value of (u, v) in the adjacent matrix that whether vertices u, v are adjacent or not in the corresponding graph. The distance measure of two graphs refers to the fraction of vertex pairs which is an edge in one graph yet not an edge in the other, taken over the domain size which is n^2. Hence, we say that an n-vertex graph is ϵ-far from a graph property $\mathcal{P}$ in the dense model if more than ϵn^2 edge insertions or removals should be performed on the graph to make the graph have the property. Otherwise, the graph G is ϵ-close to $\mathcal{P}$.

The sparse model. In the sparse model, which is suitable for sparse graphs particularly, *adjacency-lists* are commonly used. The maximum degree of a graph in this model is assumed to be bounded by d. In this model, a query of a property tester is like the question that "who is the ith neighbor of vertex v in the graph?" A null symbol $\varnothing$ is returned if there is no such neighbor of v. A property tester can probe the adjacency list of the vertices in the graph, where the maximum degree of the graph is assumed to be bounded (say, at most d). Here the distance measure of two graphs refers to the fraction of vertex pairs which is an edge in one graph yet not an edge in the other, taken over the domain size which is dn. Hence, we say that an n-vertex graph G is ϵ-far from satisfying a graph property $\mathcal{P}$ in the sparse model if more than ϵdn edge insertions and removals should be performed to make G satisfy the property. Otherwise, the graph G is ϵ-close to $\mathcal{P}$.

Graph property testing in the dense model is well understood. A large number of graph properties are shown to be testable in the dense model (see [8–11, 67, 74, 106]). However, on the other hand, the current understanding of graph property testing in the sparse model is relatively limited. To our knowledge, current known testable graph properties in the sparse model include Eulerian [76], cycle-freeness [76], connectivity [76, 118]), minor-closed properties [21, 83], hereditary properties of nonexpanding graphs [53], and properties of hyperfinite graphs [95]. There are still many graph properties which are neither testable nor known to be testable in the sparse model. From this point of view, it is worth working on devising parameterized property testers in the sparse model to see whether parameterization helps in the testing. Due to the above reasons, in the rest of this chapter we focus on graph property testing in the sparse model, and we consider the graph properties which correspond to **NP**-complete problems.

Note that there are properties which are *trivial to test* in the setting of parameterized property testing when the associated parameters k's are small and the size of the vertex set of the input graph is sufficiently large, even their corresponding parameterized problems are not in **FPT** (unless **NP** = **P**). Here we say that a graph property is trivial to test if either one can simply answer "yes" or "no" for any input graph without observing it. For example, consider the following properties.

- **The property of having a simple k-path and the property of having a simple k-cycle.** A simple k-path is a simple path of length $k-1$ and a simple k-cycle is a simple cycle of length k. To deterministically decide if a graph has a simple k-path (resp., a simple k-cycle) is **NP**-complete since it is equivalent to the notorious *Hamiltonian Path* problem (resp., *Hamiltonian Cycle* problem) when $k = n$, and it is fixed-parameter tractable [88]. Clearly, one can add at most $k-1$ edges in the graph to make it have a simple k-path. Any graph is ϵ-close to satisfying this property in the sparse model since $k-1 = o(n)$ for a constant integer k. Thus, for testing this property in the sparse model, one can simply answer "yes" for any input graph since it can never be ϵ-far from having a simple k-path. Similarly, one can simply answer "yes" for any input graph for testing if a graph has a cycle of length k. These two properties are both trivial to test.

- **The property of having a dominating set of size bounded by k.** Given a graph $G = (V, E)$, a dominating set is a subset $V' \subseteq V$ of vertices such that every vertex of G is either in V' or adjacent to at least one vertex in V'. The *Dominating Set* problem asks if a graph has a dominating set of size at most k. It is well-known to be **NP**-complete [72], and not in **FPT** [98]. In the sparse model, it is proved that testing if a graph has a dominating set of size at most ρn, for $0 < \rho < 1$, requires $\Omega(\sqrt{n})$ time [76]. However, for a constant k, the property of having a dominating set of size bounded by k is trivial to test due to the following reason. Suppose there is a graph G which has a dominating set of size k. Since a vertex is adjacent to at most d vertices in the graph, we derive that $n \leq k \cdot d + k$. Thus, we know that any graph with sufficiently large vertex set does not satisfy this property in the sparse model. One can simply answer "no" for testing this property, hence it is trivial to test in the sparse model.

- **The property of having a clique of size k and the property of having an independent set of size k.** The *Clique* problem and the *Independent Set* problem are both well-known **NP**-complete [72] problems. They are not in **FPT** [98]. Recall that a vertex subset $S \subseteq V$ is a *clique* if each pair of vertices in S are adjacent, while S is an *independent set* if none of the pairs of vertices in S are adjacent. The Clique problem asks if there exists a clique of size k while the Independent Set problem asks if there exists an independent set of size k. Clearly, the Clique problem is equivalent to the Independent Set problem in the complement graph. For any graph in the sparse model, one can add (resp., remove) $O(k^2) = o(n)$ edges to make it have a clique (resp., an independent set) of size k. Hence, any graph is ϵ-close to having a clique of size k (resp., having an independent set of size k). Hence, the properties of having a clique of size k and having an independent set of size k are both trivial to test since can simply answer "yes" for any input graph.

In the following sections, we focus on the property of *having a vertex cover of size at most* k and the property of *having a treewidth at most* k in the sparse model. Both the Vertex Cover problem and to determine if the treewidth of a graph is at most k are well-known to be fixed-parameter tractable [98]. They both admit $O(f(k) \cdot n)$ fixed-parameter algorithms, which are very efficient since they are linearly solvable with respect to n when k is a small integer. We show that their corresponding graph properties both admit efficient parameterized property testers.

5.1 Testing If a Graph Has a Vertex Cover of Size at Most k

Given a graph $G = (V, E)$, a subset $S \subseteq V$ is called a vertex cover of a graph G if for any edge $(u, v) \in E(G)$, $\{u, v\} \cap S \neq \emptyset$. The Minimum Vertex Cover problem is to find a vertex cover of minimum size in the graph. It is well-known to be **NP**-hard. The linear time 2-approximation algorithm of Gavril (*cf.* [72]) is considered as one of the jewels of theoretical computer science. It is shown to be **NP**-hard even to approximate up to a factor of 1.3606 [59].

Given a nonnegative integer k, the *parameterized Vertex Cover* problem is to decide if a graph has a vertex cover of size at most k. There are abundant of results on design of fixed-parameter algorithms for this problem. The first fixed-parameter algorithm for the parameterized Vertex Cover problem is given by Buss and Goldsmith [38] in 1993, which runs in $O(kn + 2^k k^{2k+2})$ time. There has been an impressive list of improved algorithms for the problem since 1993 (e.g., see [16, 47–

49, 99, 101, 111]). The current best time bound is $O(1.2738^k + kn)$, which is proposed by Chen et al. [48].

Let $\mathcal{P}_{VC\leq k}$ denote the property that a graph has a vertex cover of size at most k. We denote by $G \in \mathcal{P}_{VC\leq k}$ that a graph G satisfies $\mathcal{P}_{VC\leq k}$. It is clear that $\mathcal{P}_{VC\leq k}$ is a hereditary property. By making use of the result in [3], which is an extension of Szemerédi's regularity lemma [113], Alon and Shapira [11] showed that every hereditary graph property is testable with one-sided error in the dense model, while the query complexity is only guaranteed to be a function of towers of 2's of height $O(\mathrm{poly}(1/\epsilon))$. In the sparse model, Goldreich and Ron [76] proved that it requires at least $\Omega(\sqrt{n})$ queries to test $\mathcal{P}_{VC\leq \rho n}$ for a constant $0 < \rho < 1$. Note that in [103], Parnas and Ron provided an $O(d^{\log d}/\epsilon^2)$ algorithm to distinguish the case that a graph has a vertex cover of size ρn and the case in which it is ϵ-far from having a vertex cover of size $\alpha \cdot \rho n$. Such a setting is slightly weaker than that of the standard property testing.

In Sect. 5.1.1, we present an adaptive parameterized property tester with two-sided error for $\mathcal{P}_{VC\leq k}$ in the sparse model. The tester runs in $O(d/\epsilon)$ time when $k < n/(6d)$. In Sect. 5.1.2, we present an adaptive parameterized property tester with one-sided error for $\mathcal{P}_{VC\leq k}$ in the sparse model. The tester runs in $O(kd/\epsilon)$ time when $k < \epsilon n/4$.

5.1.1 A simple parameterized property tester with two-sided error

Let $[d]$ denote the set $\{1, 2, \ldots, d\}$. A graph in the sparse model is represented by an adjacency list, which can be regarded as a function $f_G : V(G) \times [d] \mapsto V(G) \cup \varnothing$ such that $f_G(v, i) = u$ if (u, v) is the ith edge incident to v (i.e., u is the ith neighbor of v), and $f_G(v, i) = \varnothing$ if there is no such edge. Let us consider the following observation.

Observation 5.1. *In the sparse model, if a graph $G = (V, E)$ satisfies $\mathcal{P}_{VC\leq k}$, then $|E| \leq kd$. Furthermore, if G is ϵ-far from $\mathcal{P}_{VC\leq k}$, then $|E| \geq \epsilon dn$.*

Proof. Suppose that $G = (V, E)$ is a graph that has a vertex cover $C \subseteq V$ of size at most k. Since each vertex in C can cover at most d edges in the graph, the number of edges in G is at most $k \cdot d$. Let us consider the case that G is ϵ-far from $\mathcal{P}_{VC\leq k}$. If $|E| < \epsilon dn$, then removing all the edges in E results in an empty graph, which has $\emptyset$ as its vertex cover. The number of edge removals is less than ϵdn. This contradicts to the assumption that G is ϵ-far from $\mathcal{P}_{VC\leq k}$. Thus in this case, we have $|E| \geq \epsilon dn$. □

Observation 5.1 implies that a graph having a vertex cover of size k is close to be empty. Based on this observation, we obtain a simple property tester for $\mathcal{P}_{VC\leq k}$, which is called Simple-VC-Tester and listed in Algorithm 5.1.

Simple-VC-Tester(G)
/* $G = (V, E)$: a graph stored in an adjacency list */
begin
  if $k < n/(6d)$ **then**
    run the $O(1.2738^k + kn)$ fixed-parameter algorithm in [48];
  else /* $k \geq n/(6d)$ */
    repeat
      choose a vertex $v \in V$ uniformly at random;
      for $i \leftarrow 1$ to d **do**
        if $f_G(v, i) \neq \varnothing$ **then**
          return "no";
        end if
      end for
    until $2/\epsilon$ times
    return "yes";
  end if
end

Algorithm 5.1: Simple-VC-Tester: a simple property tester for $\mathcal{P}_{VC\leq k}$ in the sparse model.

Theorem 5.1. *Algorithm* Simple-VC-Tester *is an adaptive parameterized property tester with two-sided error for $\mathcal{P}_{VC\leq k}$ in the sparse model, which is weakly uniform on k. In particular, its time complexity is*

$$\begin{cases} O(d/\epsilon) & \textit{if } k < n/(6d); \\ O(1.2738^k + k^2 d) & \textit{otherwise.} \end{cases}$$

Proof. When $k \geq n/(6d)$, we have $n \leq 6kd$. Then the $O(1.2738^k + kn) = O(1.2738^k + k^2 d)$ fixed-parameter algorithm in [48] is used to deterministically decide if the input graph satisfies $\mathcal{P}_{VC\leq k}$. No mistake is made by the algorithm in this case. Next, we consider the case that $k < n/(6d)$.

Consider the case that the input graph G satisfies $\mathcal{P}_{VC\leq k}$. By Observation 5.1, the probability that the vertex v chosen at Line 5 has at least one neighbor in the graph is most $2kd/n < 1/3$. Thus, the algorithm returns "no" (at Line 8) with probability at most $1/3$. Thus, the algorithm answers "yes" with probability at least $2/3$ in this case. Consider the case that G is ϵ-far from $\mathcal{P}_{VC\leq k}$. By Observation 5.1, we derive that there are at least $\epsilon dn/d = \epsilon n$ vertices which has at least one neighbor in the graph. Thus, we obtain that the algorithm returns "yes" (at Line 12) with probability at most $(1-\epsilon)^{2/\epsilon} < e^{-2} < 1/3$. Thus, the algorithm answers "no" with probability at least $2/3$ in this case.

When $k < n/(6d)$, it is easy to see that the algorithm runs in $O(d/\epsilon)$ time. Since the algorithm could make mistakes in the case that G satisfies $\mathcal{P}_{VC\leq k}$ and the case that G is ϵ-far from $\mathcal{P}_{VC\leq k}$, it has two-sided error. It is clearly adaptive since it examines neighbors of a vertex in the adjacency list. Furthermore, it is weakly uniform since it uses two different procedures for $k < n/(6d)$ and $k \geq n/(6d)$. Therefore, the theorem is proved.

$\square$

5.1.2 A parameterized property tester with one-sided error

In the following we give a parameterized property tester with one-sided error for $\mathcal{P}_{VC\leq k}$ in the sparse model. This parameterized property tester is called VC-FPT-Tester, which is listed in Algorithm 5.2.

For $k \geq \epsilon n/4$, Algorithm VC-FPT-Tester runs the $O(1.2738^k + kn) = O(1.2738^k + k^2/\epsilon)$ fixed-parameter algorithm to deterministically decide if the input graph satisfies $\mathcal{P}_{VC\leq k}$, hence its correctness and complexity is clear for such k's. In the following, we consider the case that $k < \epsilon n/4$, and we prove that Algorithm VC-FPT-Tester satisfies the following two constraints.

- VC-FPT-Tester returns "yes" if G satisfies $\mathcal{P}_{VC\leq k}$;
- VC-FPT-Tester returns "no" with probability at least $2/3$ if G is ϵ-far from $\mathcal{P}_{VC\leq k}$.

First, we consider the case that $G \in \mathcal{P}_{VC\leq k}$. The algorithm tries to find a matching of G by looking for a set of disjoint edges (i.e., a set of edges $E' \subseteq E$ such that for every two edges $(u,v),(u',v') \in E'$, $\{u,v\} \cap \{u',v'\} = \emptyset$). Note that the

VC-FPT-Tester(G, k)
/* $G = (V, E)$: a graph stored in an adjacency list with bounded degree d;
  k: an integer parameter */
begin
1: **if** $k \geq \epsilon n/4$ **then**
2:   run the $O(1.2738^k + kn)$ fixed-parameter algorithm in [48];
3: **else** /* $k < \epsilon n/4$ */
4:   $t \leftarrow 0$;
5:   **repeat**
6:     choose a vertex $v \in V$ uniformly at random;
7:     **if** v is marked **then continue**;
8:     **for** $i \leftarrow 1$ to d **do**
9:       **if** $f_G(v, i) \neq \varnothing$, and $f_G(v, i)$ is not marked **then**
10:         $t \leftarrow t + 1$;
11:         mark v and $f_G(v, i)$;
12:         **break**; /* Exit the for-loop */
13:       **end if**
14:     **end for**
15:   **until** $\lceil 10k/\epsilon \rceil$ times
16:   **return** "no" if $t \geq k + 1$, otherwise **return** "yes".
17: **end if**
end

Algorithm 5.2: VC-FPT-Tester: a parameterized property tester for $\mathcal{P}_{VC \leq k}$ in the sparse model.

size of a matching is always smaller than or equal to the size of a vertex cover in the graph since any vertex cover must contain at least one endpoint of each matched edge. Based on this observation, the algorithm never returns "no" in this case.

Next, let us consider the case that G is ϵ-far from satisfying $\mathcal{P}_{VC \leq k}$. In this case, it is clear that $|E(G)| \geq \epsilon dn$. Let A_i be the number of finished iterations of the loop (in Lines 5–15) such that i disjoint edges are found. Let $X_i = A_i - A_{i-1}$, hence $A_i = \sum_{j=0}^{i} X_j$ where $X_0 = A_0 = 0$. Let Y_i be the event that a new edge is found whose endpoints are not in the previous found $i - 1$ disjoint edges. Since there are at least $\epsilon dn/d = \epsilon n$ vertices of degree greater than 0, we have $\mathbf{Pr}[Y_1] \geq \epsilon n/n = \epsilon$. Similarly, we obtain that $\mathbf{Pr}[Y_i] \geq (\epsilon dn - 2(i-1)d)/dn = \epsilon - 2(i-1)/n$. Thus, the expected value of the geometric random variable X_i is $\mathbf{E}[X_i] \leq 1/(\epsilon - 2(i-1)/n)$, and then we have

$$\begin{aligned}
\mathbf{E}[A_{k+1}] &\le \frac{1}{\epsilon} + \frac{1}{\epsilon - 2/n} + \ldots + \frac{1}{\epsilon - 2k/n} \\
&\le \frac{1}{\epsilon} + \frac{k}{\epsilon - 2k/n} \\
&< \frac{1}{\epsilon} + \frac{k}{\epsilon - \epsilon/2} \quad (\because k < \epsilon n/4) \\
&\le \frac{2k+1}{\epsilon}.
\end{aligned}$$

Thus, the probability that Algorithm VC-FPT-Tester returns "yes" in this case is

$$\mathbf{Pr}\left[A_{k+1} > \left\lceil \frac{10k}{\epsilon} \right\rceil\right] \le \mathbf{Pr}\left[A_{k+1} \ge \frac{10k}{\epsilon}\right] \le \frac{(2k+1)/\epsilon}{10k/\epsilon} \le \frac{3k}{10k} < \frac{1}{3},$$

where the second inequality follows by Markov's inequality.

As the time complexity of the algorithm depends on the number of queries performed to seek for disjoint edges, we have that Algorithm VC-FPT-Tester runs in $O(kd/\epsilon)$ time. It is easy to see, just like Algorithm Simple-VC-Tester, that Algorithm VC-FPT-Tester is adaptive and weakly uniform on k. Furthermore, it never makes mistakes for the case that G satisfies $\mathcal{P}_{VC\le k}$. Therefore, Theorem 5.2 immediately follows.

Theorem 5.2. *Algorithm* VC-FPT-Tester *is an adaptive parameterized property tester with one-sided error for $\mathcal{P}_{VC\le k}$ in the sparse model, which is weakly uniform on k. In particular, its time complexity is*

$$\begin{cases} O(kd/\epsilon) & \textit{if } k < \epsilon n/4; \\ O(1.2738^k + k^2/\epsilon) & \textit{otherwise.} \end{cases}$$

Remarks. In fact, Algorithm VC-FPT-Tester can be slightly modified so that we can obtain a parameterized property tester for $\mathcal{P}_{VC\le k}$ in the *dense* model. However, a graph satisfying $\mathcal{P}_{VC\le k}$ is sparse for when k is small since it must have less than kn edges. Thus, the sparse model is more suitable for the testing for $\mathcal{P}_{VC\le k}$.

5.2 Testing If a Graph Has Treewidth at Most k

The treewidth of a graph is one of the most important invariants of graphs. This notion was introduced by Robertson and Seymour as part of their proof of the Graph Minor Theorem [107]. Treewidth measures how close a graph is to being a tree. It now plays an important role in algorithmic graph theory, and in particular, has a large number of applications in fixed-parameter algorithms for parameterized graph problems. Many graph problems can be solved in polynomial time or even linear time when the treewidth of the input graph is bounded. Graphs with treewidth at most k are also known as *partial k-trees* [87]. A k-tree is a graph defined recursively as follows. A clique is a k-tree. For a graph $G = (V, E)$ which is a k-tree, adding a new vertex v to G and making it adjacent to exactly all vertices of a clique of size k in G form a new k-tree. Any subgraph of a k-tree is called a partial k-tree. See [60] for more details and [32] for the survey on algorithmic results on determining the treewidth of a graph.

For an integer $k > 0$, the property of having treewidth at most k is a minor-closed graph property [87]. That is, every minor of a graph with treewidth at most k also has treewidth at most k. To determine whether the treewidth of a graph is at most k is **NP**-complete [13], even for graphs with maximum degree bounded by 9 [30]. Robertson and Seymour [105] proved that this problem is in **FPT** [105]. By Alon and Shapira's result in [11], it is clear that the property of having treewidth at most k is testable with one-sided error in the dense model. Since a graph $G = (V, E)$ of treewidth at most k has $o(n^2)$ edges (see Fact 5.1), the sparse model is more suitable than the dense model for the testing of $\mathcal{P}_{tw\leq k}$. Hence, we focus on the testing of this property in the sparse model. In [21], it is shown that for every (finite) graph H, the property of being H-minor free is testable in the sparse model. In one of the deepest results in graph theory, Robertson and Seymour proved the famous Graph Minor Theorem [107], which states that there is a finite family of graphs $\mathcal{H}_{\mathcal{P}}$ such that a graph satisfies $\mathcal{P}$ if and only if it is H-minor free for all $H \in \mathcal{H}_{\mathcal{P}}$. The set of graphs $\mathcal{H}_{\mathcal{P}}$ is called *the set of forbidden minors* of $\mathcal{P}$. Follows this immediately, every minor-closed graph property is testable, however, the running time of the property tester in [21] is $O(2^{2^{2^{\mathrm{poly}(1/\epsilon)}}})$, and the analysis is quite complicated. Using the locality lemma given in [97], Hassidim et al. [83] simplified the proof in [21] and claimed a better time bound on testing minor-closed properties, which is $O(2^{\mathrm{poly}(1/\epsilon)})$.

Unfortunately, if the set $\mathcal{H}_{\mathcal{P}}$ of forbidden graph minors of property $\mathcal{P}$ is not explicitly known, then one does not know how to test property $\mathcal{P}$ using their results. In particular, we do not know the set of forbidden minors of the class of graphs with treewidth at most k for $k > 3$ [87]. We denote by $\mathcal{P}_{tw\leq k}$ the property of having treewidth at most k. In this section, we show how to test whether a graph belongs to $\mathcal{P}_{tw\leq k}$ in the sparse model by giving a parameterized property tester. We utilize the approach in [83] without knowing the set of forbidden graph minors of $\mathcal{P}_{tw\leq k}$ in advance. Our parameterized property testers for $\mathcal{P}_{tw\leq k}$ are *uniform* on the parameter k. Our first parameterized property tester for $\mathcal{P}_{tw\leq k}$ has time complexity $2^{d^{O(kd^3/\epsilon^2)}}$. By applying the concept of the local distributed partitioning oracle in [102], we obtain another parameterized property tester for $\mathcal{P}_{tw\leq k}$, which runs in time $d^{(k/\epsilon)^{O(k^2)}} + 2^{\mathrm{poly}(k,d,1/\epsilon)}$.

5.2.1 Preliminaries

Definition 5.1. Let $G = (V, E)$ be a graph. A vertex subset $I \subseteq V$ is called a (δ, α)-*nonexpanding set* if the following conditions are satisfied:

1. $G[I]$ is connected;
2. $\frac{|N_G(I)|}{|I|} \leq \delta$;
3. $|I| \leq \alpha$.

Definition 5.2 (Tree-decomposition [87])**.** A tree-decomposition of a graph $G = (V, E)$ is a pair $(\mathcal{S}, \mathcal{T})$ with $\mathcal{S} = \{\mathcal{X}_i \mid i \in \mathcal{I}\}$ a collection of subsets of vertices of G and $\mathcal{T}$ a tree where each node is associated with one subset in $\mathcal{S}$, such that the following three conditions are satisfied:

1. $\bigcup_{i\in\mathcal{I}} \mathcal{X}_i = V$;
2. for all edges $(v, w) \in E$, there is a subset $\mathcal{X}_i \in \mathcal{S}$ such that both v and w are contained in $\mathcal{X}_i$;

3. for each vertex x, the set of nodes $\{X_i \in \mathcal{S} \mid x \in \mathcal{X}_i\}$ forms a subtree of $\mathcal{T}$.

We call $\max_{i \in \mathcal{I}}\{|\mathcal{X}_i| - 1\}$ the *width* of the tree-decomposition $(\mathcal{S}, \mathcal{T})$. The *treewidth* of G, denoted by $tw(G)$, is the minimum width over all tree-decompositions of G. Furthermore, if $\mathcal{T}$ is a rooted tree, then $(\mathcal{S}, \mathcal{T})$ is called a *rooted tree decomposition* of G.

Figure 5.1 illustrates a rooted tree-decomposition of a graph. With a slight abuse of notation, for a tree-decomposition $(\mathcal{S}, \mathcal{T})$ of a graph, we use $\{\mathcal{X}_i \mid i \in \mathcal{I}\}$ to denote the set of nodes in $\mathcal{T}$.

Remark. Since graphs with maximum degree $d \leq 1$ have treewidth at most one so that the testing becomes trivial, we assume that the maximum degree d of the input graph is as least two.

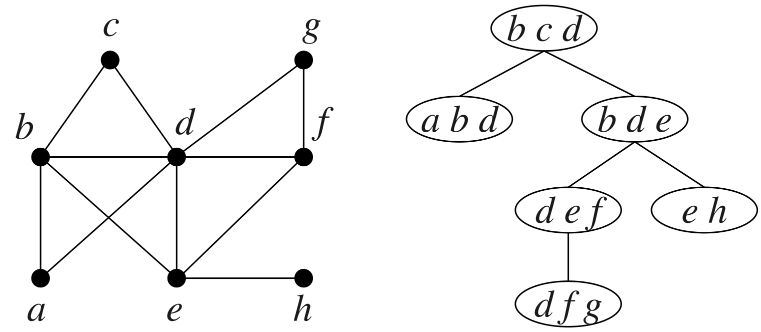

Figure 5.1: A graph with and one of its rooted tree-decompositions.

Definition 5.3 (Nice tree-decomposition [87])**.** A nice tree-decomposition $(\mathcal{S}, \mathcal{T})$ of a graph $G = (V, E)$ is a rooted tree-decomposition of G with the following conditions:

1. every node of $\mathcal{T}$ has at most two children;

2. if a node $\mathcal{X}_i$ has two children $\mathcal{X}_j$ and $\mathcal{X}_k$, then $\mathcal{X}_i = \mathcal{X}_j = \mathcal{X}_k$;

3. if a node $\mathcal{X}_i$ has only one child $\mathcal{X}_j$, then either $|\mathcal{X}_i| = |\mathcal{X}_j| + 1$ and $\mathcal{X}_j \subset \mathcal{X}_i$ or $|\mathcal{X}_i| = |\mathcal{X}_j| - 1$ and $\mathcal{X}_i \subset \mathcal{X}_j$.

The rooted tree-decomposition in Figure 5.2 is a nice tree-decomposition of the graph in left side of Figure 5.1.

Lemma 5.1 ([87]). *Every graph G with treewidth k has a nice tree-decomposition of width k.*

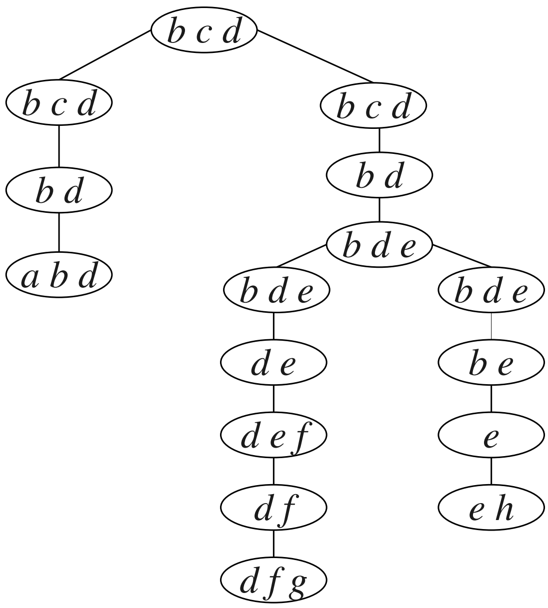

Figure 5.2: A nice tree-decomposition of the graph in Figure 5.1.

Lemma 5.2 ([57]). *If a graph has treewidth at most k, then every minor of G has treewidth at most k.*

5.2.2 Partitioning the graph into small connected components

Proposition 5.1 ([65]). *Let $G = (V, E)$ be an n-vertex graph with $\Delta(G) \leq d$ and $tw(G) \leq k$. Then for every integer $\epsilon \leq 1/2$ there exists a set $U \subseteq V$ such that $|U| \leq \epsilon n$ and $G - U$ contains no simple path with L edges, where $L = \lceil (d(d+1)(9k+7)-1)/2 \rceil^{2/\epsilon}$.*

Proposition 5.1 guarantees that for any graph with vertex degree bounded by d and treewidth bounded by k, there exists a subset $U \subseteq V$ of size at most ϵn such that removing U from the graph G results in connected components of size bounded by $d^{\lceil L/2 \rceil}$, where $L = \lceil (d(d+1)(9k+7)-1)/2 \rceil^{2/\epsilon}$. Using the nice tree-decomposition of a graph G with treewidth bounded by k, we improve Proposition 5.1 by giving a much smaller upper bound on the size of such connected components derived by removing U. This result is presented in Proposition 5.2 as follows.

Proposition 5.2. *Let G be an n-vertex graph with $tw(G) \leq k$, then for any $0 < \epsilon < 1$, there is a set $U \subseteq V$ such that $|U| \leq \epsilon n$ and $G - U$ has connected components of size at most $2(k+1)/\epsilon$.*

Proof. Let $(\mathcal{S}, \mathcal{T})$ be a nice tree-decomposition of width k of G. Let $\mathcal{R}$ be the root of $\mathcal{T}$. For each node $\mathcal{X}$ in $\mathcal{T}$, we denote by $\mathcal{X}_\ell$ and $\mathcal{X}_r$ the left child and the right child, respectively, of $\mathcal{X}$. If $\mathcal{X}$ has only one child, then we let $\mathcal{X}_\ell = \mathcal{X}_r$. We denote by $\mathcal{T}_\mathcal{X}$ the subtree of $\mathcal{T}$ rooted at $\mathcal{X}$. Let $\mathcal{S}_{\mathcal{T}_\mathcal{X}} \subseteq \mathcal{S}$ be the set of nodes in $\mathcal{T}_\mathcal{X}$. We define $\psi_\mathcal{T}(\mathcal{X}) = \bigcup_{\mathcal{Y} \in \mathcal{S}_{\mathcal{T}_\mathcal{X}}} \mathcal{Y}$ to be the set of vertices in the subsets corresponding to the nodes in $\mathcal{S}_{\mathcal{T}_\mathcal{X}}$. Consider the following algorithm for constructing the set $U \subseteq V$ as claimed in the proposition. The algorithm repeatedly runs until the graph G becomes empty. In each round of the algorithm, it starts by visiting the root $\mathcal{R}$. Whenever a node $\mathcal{X}$ is visited, the algorithm computes $\psi_\mathcal{T}(\mathcal{X})$. If $|\psi_\mathcal{T}(\mathcal{X})| > 2(k+1)/\epsilon$, then the algorithm computes $\mathcal{X}' = \operatorname{argmax}\{|\psi_\mathcal{T}(\mathcal{X}_\ell)|, |\psi_\mathcal{T}(\mathcal{X}_r)|\}$, and turns to visit $\mathcal{X}'$ in the next round. Otherwise, the algorithm stops visiting nodes after $\mathcal{X}$ is visited. Then it adds the vertices in $\mathcal{X}$ into U and removes the vertices in $\psi_\mathcal{T}(\mathcal{X})$ from the graph G. Denote by G' the resulting graph. The algorithm computes a nice tree-decomposition of G' and continues the next round.

In each round of the algorithm, it stops when a node $\mathcal{X}$ with $|\psi_\mathcal{T}(\mathcal{X})| \leq 2(k+1)/\epsilon$ is visited. Let $\mathcal{Y}$ be the parent node of $\mathcal{X}$ in $\mathcal{T}$. Here we claim that $|\psi_\mathcal{T}(\mathcal{X})| \geq (k+1)/\epsilon$. Assume the contrary that $|\psi_\mathcal{T}(\mathcal{X})| < (k+1)/\epsilon$. If $\mathcal{Y}$ has two children, say $\mathcal{X}$ and $\mathcal{Z}$, then by Condition 2 in Definition 5.3 we know $\mathcal{X} = \mathcal{Y} = \mathcal{Z}$, hence we have

$$|\psi_\mathcal{T}(\mathcal{Y})| = |\psi_\mathcal{T}(\mathcal{X}) \cup \psi_\mathcal{T}(\mathcal{Z}) \cup \mathcal{Y}| = |\psi_\mathcal{T}(\mathcal{X}) \cup \psi_\mathcal{T}(\mathcal{Z})| < \frac{2(k+1)}{\epsilon}.$$

If $\mathcal{Y}$ has only one child (i.e., $\mathcal{X}$), then by Condition 3 in Definition 5.3 we know that either $\mathcal{X} \subset \mathcal{Y}$ and $|\mathcal{Y}| = |\mathcal{X}| + 1$ or $\mathcal{Y} \subset \mathcal{X}$ and $|\mathcal{Y}| = |\mathcal{X}| - 1$. Hence

$$|\psi_\mathcal{T}(\mathcal{Y})| \leq |\psi_\mathcal{T}(\mathcal{X})| + 1 < \frac{k+1}{\epsilon} + 1 \leq \frac{2(k+1)}{\epsilon}.$$

By these two cases we obtain that $|\psi_\mathcal{T}(\mathcal{Y})| < 2(k+1)/\epsilon$, which implies that the algorithm stops visiting nodes at $\mathcal{Y}$ before $\mathcal{X}$ so that a contradiction occurs. Note that as the vertices in $\mathcal{X}$ are removed from the graph, we obtain induced subgraphs $G[\psi_\mathcal{T}(\mathcal{X}_\ell)]$, $G[\psi_\mathcal{T}(\mathcal{X}_r)]$, and $G[V \setminus (\psi_\mathcal{T}(\mathcal{X}_\ell) \cup \psi_\mathcal{T}(\mathcal{X}_r))]$ with no edge between them (by

Condition 3 in Definition 5.2). Both $\psi_{\mathcal{T}}(\mathcal{X}_\ell)$ and $\psi_{\mathcal{T}}(\mathcal{X}_r)$ are of size at most $2(k+1)/\epsilon$. Thus, as the algorithm terminates, we obtain connected components of G each of which is of size at most $2(k+1)/\epsilon$. Since $tw(G) \le k$ and $tw(G') \le k$ for every induced subgraph G' of G (by Lemma 5.2), each subset $\mathcal{X}$ with its vertices added into U is of size at most $k+1$. Moreover, the algorithm removes at least $(k+1)/\epsilon$ vertices from the graph in each round. Thus, the size of U is at most

$$\frac{n}{(k+1)/\epsilon} \cdot (k+1) = \epsilon n.$$

Therefore, the proposition is proved. □

Lemma 5.3. *Let $G = (V, E)$ be an n-vertex graph with $\Delta(G) \le d$ and $tw(G) \le k$. Let G' be an induced subgraph of G. Then for any $\epsilon \in (0, 1)$ and $\beta > 1$, the probability that a vertex chosen uniformly at random in $G' = (V', E')$ is not contained in any $(\epsilon/\beta, \zeta(k, d, \epsilon))$-nonexpanding set is at most ϵ/β where $\zeta(k, d, \epsilon) = 4\beta^2 d(k+1)/\epsilon^2$.*

Proof. Since $tw(G) \le k$, any induced subgraph $G' = (V', E')$ of G has treewidth at most k. By Proposition 5.2 we know that, for any $0 < \epsilon' < 1$, there exists a set $U' \subset V'$, $|U'| \le \epsilon'|V'|$ such that every connected component of $G'[V' \setminus U']$ has at most $2(k+1)/\epsilon'$ vertices. Let $\mathcal{C}$ be the collection of connected components of $G'[V' \setminus U']$. For any $C \in \mathcal{C}$, we define $\gamma(v) = |N_{G'}(C)|/|C|$ for each $v \in C$. For $v \in U'$, we define $\gamma(v) = d$. Note that $\bigcup_{C \in \mathcal{C}} N_{G'}(C) = U'$. Furthermore, since $\Delta(G') \le d$, a vertex in U' is adjacent to at most d different connected components in $\mathcal{C}$ so that we have $\sum_{C \in \mathcal{C}} |N_{G'}(C)| \le d|U'|$. Thus, for a vertex v picked uniformly at random from G', we have

$$\begin{aligned}
\mathbf{E}_{v \in V'}[\gamma(v)] &= \sum_{C \in \mathcal{C}} \sum_{v \in C} \mathbf{Pr}[v \text{ is picked}] \cdot \frac{|N_{G'}(C)|}{|C|} + \sum_{v \in U} \mathbf{Pr}[v \text{ is picked}] \cdot d \\
&= \sum_{C \in \mathcal{C}} \sum_{v \in C} \frac{1}{|V'|} \cdot \frac{|N_{G'}(C)|}{|C|} + \sum_{v \in U} \frac{1}{|V'|} \cdot d \\
&= \sum_{C \in \mathcal{C}} |C| \cdot \frac{1}{|V'|} \cdot \frac{|N_{G'}(C)|}{|C|} + |U'| \cdot \frac{1}{|V'|} \cdot d \\
&= \frac{1}{|V'|} \cdot \sum_{C \in \mathcal{C}} |N_{G'}(C)| + d \cdot \frac{|U'|}{|V'|} \\
&\le \frac{2d \cdot |U'|}{|V'|} \\
&\le 2d\epsilon'.
\end{aligned}$$

Take $\epsilon' = \epsilon^2/(2\beta^2 d)$ for any $\epsilon \in (0,1)$ and any $\beta > 1$, we have

$$\mathbf{E}_{v\in V'}[\gamma(v)] \le \frac{\epsilon^2}{\beta^2}.$$

Each connected component in $\mathcal{C}$ is of size at most

$$\zeta(k,d,\epsilon) = \frac{2(k+1)\cdot 2\beta^2 d}{\epsilon^2} = \frac{4\beta^2 d(k+1)}{\epsilon^2}.$$

Similar to the definition of an $(\epsilon/\beta, \zeta(k,d,\epsilon))$-nonexpanding set, we call a connected component $C \in \mathcal{C}$ an $(\epsilon/\beta, \zeta(k,d,\epsilon))$-*nonexpanding component* if $|N_{G'}(C)|/|C| \le \epsilon/\beta$. By Markov's inequality, the probability that a vertex v chosen uniformly at random in V' with $\gamma(v) \ge \epsilon/\beta$ is at most ϵ/β, so the probability that a vertex is not contained in any $(\epsilon/\beta, \zeta(k,d,\epsilon))$-nonexpanding component of $G[V' \setminus U']$ is at most ϵ/β, that is, the probability that a vertex is contained in an $(\epsilon/\beta, \zeta(k,d,\epsilon))$-nonexpanding component of $G[V' \setminus U']$ is larger than $1-\epsilon/\beta$. Since the set of $(\epsilon/\beta, \zeta(k,d,\epsilon))$-nonexpanding components of G' includes the $(\epsilon/\beta, \zeta(k,d,\epsilon))$-nonexpanding components of $G[V' \setminus U']$, the probability that a vertex chosen uniformly at random in G' is contained in an $(\epsilon/\beta, \zeta(k,d,\epsilon))$-nonexpanding set is larger than $1-\epsilon/\beta$. Therefore, the lemma follows. □

We abbreviate an $(\epsilon/\beta, \zeta(k,d,\epsilon))$-nonexpanding component to a nonexpanding component if the context is clear.

Lemma 5.4. *Let $G = (V,E)$ be the input of Algorithm* Global-Partition *with $\Delta(G) \le d$ and $tw(G) \le k$. Then for any $\epsilon \in (0,1)$ and $\beta > 1$, by setting parameters $\delta = \epsilon/\beta$ and $\alpha = \zeta(k,d,\epsilon)$, Algorithm* Global-Partition *returns a vertex set U whose expected size is at most $2\epsilon dn/\beta$ and the probability that $|U| \le \epsilon n/4$ is at least $1 - 8d/\beta$.*

***Proof*.** For a graph $G = (V,E)$ by setting $\Delta(G) \le d$ and $tw(G) \le k$, Algorithm Global-Partition partitions V into sets of size at most $\zeta(k,d,\epsilon)$ with $\delta = \epsilon/\beta$ and $\alpha = \zeta(k,d,\epsilon)$. We define a sequence of random variables X_i, $1 \le i \le n$, as follows. X_i corresponds to the ith vertex removed by Algorithm Global-Partition from the graph. Say, the remaining graph has $n-h$ vertices, and the algorithm is removing a set $I \cup S = \{v_{h+1}, \ldots, v_{h+y}\}$ of y vertices. Then for $h+1 \le j \le h+y$, we set $X_j = |S|/|I|$ if $v_j \in I$ and $X_j = 0$ if $v_j \in S$. Note that $\sum_{i=1}^{n} X_i$ equals the number of vertices in U. Consider the following three cases:

```
Global-Partition(G, δ, α)
/* G = (V, E): a graph stored in an adjacency list with Δ(G) ≤ d;
   δ, α: the arguments of the nonexpanding sets */
begin
 1: (π_1, ..., π_n) ← random permutation of vertices in V;
 2: U ← ∅; P ← ∅;
 3: for i ← 1 to n do
 4:    if π_i is still in the graph then
 5:       if there exists a (δ, α)-nonexpanding set I in G that contains π_i then
 6:          S ← N_G(I);
 7:       else
 8:          I ← {π_i}; S ← N_G(π_i);
 9:       end if
10:       U ← U ∪ S; P ← P ∪ {(I ∪ S)};
11:       remove vertices in I ∪ S from G;
12:    end if
13: end for
end
```

Algorithm 5.3: Global-Partition: the global partitioning algorithm.

(i) v_i is not contained in any nonexpanding set of G;

(ii) v_i is contained in some nonexpanding set of G.

Case (i) occurs with probability at most ϵ/β by Lemma 5.3. By Line 8 of the algorithm, we derive that $X_i \le d$ in this case. Note that in this case if $v_i \in N_G(I)$ for some nonexpanding set I of G, then by definition we have $X_i = 0$. As for case (ii), it is clear that $X_i \le \delta = \epsilon/\beta$. Therefore, for each $1 \le i \le n$, we have

$$\mathbf{E}[X_i] \le \frac{\epsilon}{\beta} + d \cdot \frac{\epsilon}{\beta} \le \frac{2\epsilon d}{\beta},$$

and the expected number of vertices in U is $\mathbf{E}[\sum_{i=1}^{n} X_i] \le 2\epsilon dn/\beta$ by the union bound. Furthermore, Markov's inequality implies that the probability of $|U| > \epsilon n/4$ is at most $8d/\beta$. Thus, the probability of $|U| \le \epsilon n/4$ is at least $1 - 8d/\beta$. □

5.2.3 The partitioning oracle and the property tester for $\mathcal{P}_{tw\le k}$

Let $\mathcal{P}$ be the partition obtained by Algorithm Global-Partition with $\delta = \epsilon/\beta$ and $\alpha = \zeta(k, d, \epsilon)$. Each set A in the partition $\mathcal{P}$ is the union of a set I and its open neighborhood S, which are referred as A_I and A_S respectively. We use $\mathcal{P}[v]$ to denote the set in $\mathcal{P}$ which contains v. Clearly, we have $U = \bigcup_{v \in V} (\mathcal{P}[v])_S$.

Definition 5.4 ([83])**.** We say that $\mathcal{O}$ is a (τ, ω)-*partitioning oracle* for a graph class $\mathcal{G}$ if given query access to a graph $G = (V, E)$ in the adjacency-list model, it provides query access to a partition $\mathcal{P}$ of V such that for a query about $v \in V$, $\mathcal{O}$ either returns $\mathcal{P}[v]_I$ or answers that $v \in U$. Furthermore, the partition $\mathcal{P}$ has the following properties:

- $\mathcal{P}$ is a function of the graph and random bits of the oracle. In particular, it does not depend on the order of queries to $\mathcal{O}$.
- For every $v \in V$, $|(\mathcal{P}[v])_I| \leq \omega$ and $G[(\mathcal{P}[v])_I]$ is connected.
- If $G \in \mathcal{G}$, then $|U| \leq \tau n$ with probability at least $\frac{82}{90}$.

Lemma 5.5. *For any $\epsilon \in (0, 1)$, there is an $(\epsilon/4, \zeta(k, d, \epsilon))$-partitioning oracle for the graph class $\mathcal{P}_{tw \leq k}$, which consists of graphs $G = (V, E)$ with $\Delta(G) \leq d$ and $tw(G) \leq k$, where $\zeta(k, d, \epsilon) = 4\beta^2 d(k + 1)/\epsilon^2$ and $\beta = 90d$.*

Proof. By Lemma 5.4, with parameters $\delta = \epsilon/\beta$, $\alpha = \zeta(k, d, \epsilon) = 4\beta^2 d(k + 1)/\epsilon^2$, and $\beta = 90d$, Algorithm Global-Partition computes connected components of $G - U$ of size at most $\zeta(k, d, \epsilon)$. Moreover, the probability that Algorithm Global-Partition returns set U of size at most $\epsilon n/4$ is at least $1 - 8/90$. Hence, Algorithm Global-Partition is an $(\epsilon/4, \zeta(k, d, \epsilon))$-partitioning oracle for the graph class $\mathcal{P}_{tw \leq k}$. The lemma is then proved. □

Define by $B_G(v, r) = \{u \in V(G) \mid d(u, v) \leq r\}$ the set of vertices in G of distance at most r from v. Our property tester for testing $tw(G) \leq k$ is given as Algorithm Treewidth-Tester, where $\mathcal{O}$ is an $(\epsilon/4, \zeta(k, d, \epsilon))$-partitioning oracle for the graph class $\mathcal{P}_{tw \leq k}$. Be noted that we do not care about the complexity of constructing the oracle $\mathcal{O}$ for the moment. In the next subsection, we shall present how to simulate $\mathcal{O}$ in time independent of n.

We say that the set U obtained by the partitioning oracle $\mathcal{O}$ of Algorithm Treewidth-Tester is a *helpful dividing set* if $|U| \leq \epsilon n/4$. It is easy to see that for any graph $G = (V, E)$ with $\Delta(G) \leq d$ and $tw(G) \leq k$, Lemma 5.5 implies that there exists an $(\epsilon/4, \zeta(k, d, \epsilon))$-partitioning oracle which derives a helpful U.

Treewidth-Tester$(G, \mathcal{O}, k)$
/* $G = (V, E)$: a graph stored in an adjacency list with $\Delta(G) \leq d$;
$\mathcal{O}$: an $(\epsilon/4, \zeta(k, d, \epsilon))$-partitioning oracle for $\mathcal{P}_{tw \leq k}$;
k: an integer parameter. */
begin
/* Stage I: */
$f \leftarrow 0$;
for $j \leftarrow 1$ to t_1 **do**
  pick a vertex $v \in V$ uniformly at random;
  if $\mathcal{O}$ says $v \in U$ **then**
    $f \leftarrow f + 1$;
  end if
end for
if $f/t_1 \geq 3\epsilon/8$ **then**
  return "no";
end if
/* Stage II: */
select independently and uniformly at random a set $S \subset V$ of size t_2;
if $G[\bigcup_{s \in S} B_G(s, \zeta(k, d, \epsilon) - 1)]$ has treewidth greater than k **then**
  return "no";
else
  return "yes";
end if
end

Algorithm 5.4: Treewidth-Tester: a property tester for testing $\mathcal{P}_{tw \leq k}$ in the sparse model.

Lemma 5.6. *Let $G = (V, E)$ be a graph with $\Delta(G) \leq d$. If the set U computed by the $(\epsilon/4, \zeta(k, d, \epsilon))$-partitioning oracle of Algorithm* Treewidth-Tester *is a helpful dividing set and $t_1 = 256/\epsilon^2$, then the probability that $f/t_1 \geq \epsilon/2$ (in other words, $f \geq t_1\epsilon/2$) is at most* $1/1000$.

Proof. Let χ_v^U be an indicator random variable such that, for $v \in V$, $\chi_v^U = 1$ if $v \in U$ and $\chi_v^U = 0$ otherwise. Then $f = \sum_{v \in S} \chi_v^U$ denotes the sum of the indicator random variables χ_v^U for $v \in S$. By the assumption that U is a helpful dividing set, we have $\mathbf{Pr}[\chi_v^U = 1] \leq \frac{\epsilon}{4}$. Let μ denote $\mathbf{E}[f]$. Hence, we have

$$\mu = \mathbf{E}\left[\sum_{v \in S} \chi_v^U\right] \leq \frac{\epsilon}{4} \cdot |S| = \frac{\epsilon}{4} \cdot \frac{256}{\epsilon^2} = \frac{64}{\epsilon}.$$

Thus by the Chernoff bound, we have

$$\begin{aligned}\mathbf{Pr}\left[f \geq \frac{3\epsilon}{8}t_1\right] &= \mathbf{Pr}\left[f \geq \frac{96}{\epsilon}\right] \leq \mathbf{Pr}\left[f \geq \left(1+\frac{1}{2}\right)\cdot\mu\right] \\ &\leq \left(\frac{e^{1/2}}{(1+1/2)^{(1+1/2)}}\right)^{64/\epsilon} \\ &\leq \left(\frac{e^{1/2}}{(1+1/2)^{(1+1/2)}}\right)^{64} \\ &< 0.001.\end{aligned}$$

□

To make discussions concise, herein we say that Algorithm Treewidth-Tester accepts the input graph G if it answers "yes" and rejects G if it answers "no".

Theorem 5.3. *With $t_1 = 256/\epsilon^2$ and $t_2 = 4/\epsilon$, Algorithm* Treewidth-Tester *accepts a graph of treewidth no larger than k and rejects a graph ϵ-far from having treewidth at most k both with probability greater than $2/3$, respectively.*

Proof. Algorithm Treewidth-Tester uses the $(\epsilon/4, \zeta(k,d,\epsilon))$-partitioning oracle described in Lemma 5.5. The probability that the oracle computes a helpful dividing set U is at least $82/90$. For a graph G of $tw(G) \leq k$, Algorithm Treewidth-Tester might reject it in Step 10 if the set U is not a helpful dividing set or U is a helpful dividing set but $f/t_1 > \epsilon/2$. By Lemma 5.6, the probability that U is not a helpful dividing set but $f/t_1 > \epsilon/2$ is less than $1/1000$. Thus Algorithm Treewidth-Tester rejects G in Step 10 with probability at most $8/90 + (82/90)\cdot(1/1000) < 0.1$. Since every induced subgraph of G must have treewidth at most k, Algorithm Treewidth-Tester never rejects G in Step 15. Thus, G is accepted by Algorithm Treewidth-Tester with probability at least $9/10 > 2/3$.

Consider the case when G is ϵ-far from $tw(G) \leq k$. Algorithm Treewidth-Tester can only accept G in Step 17. We finished the proof by the following two cases that whether the set U computed by the oracle is a helpful dividing set or not.

(1) Suppose that the set U computed by the oracle is of size greater than $\epsilon n/2$. We claim that G will be rejected by Algorithm Treewidth-Tester in Step 10 with probability at least 0.86. Similar to the proof of Lemma 5.6, let z_v^U be an indicator random variable such that $z_v^U = 1$ if $v \in U$ and $z_v^U = 0$

otherwise. Then $f = \sum_{v \in S} z_v^U$ denotes the number of vertices picked in Step 4 of Algorithm Treewidth-Tester which are in U. Let μ' denote $\mathbf{E}[f]$. Since $|U| > \epsilon n/2$, we derive that $\mathbf{Pr}[z_v^U = 1] > \epsilon/2$, which implies that $\mu' > (\epsilon/2) \cdot |S| = (\epsilon/2) \cdot t_1 = (\epsilon/2) \cdot 256/\epsilon^2 = 128/\epsilon$. Hence, by the Chernoff bound we obtain that G is accepted with probability at most

$$\begin{aligned}
& \mathbf{Pr}\left[f < \frac{3\epsilon}{8} t_1\right] = \mathbf{Pr}\left[f < \frac{96}{\epsilon}\right] = \mathbf{Pr}\left[f < \frac{3}{4} \cdot \frac{128}{\epsilon}\right] \\
\leq \; & \mathbf{Pr}\left[f \leq \left(1 - \frac{1}{4}\right)\mu'\right] \\
\leq \; & e^{-\mu' \cdot \left(\frac{1}{4}\right)^2 \cdot \frac{1}{2}} \\
\leq \; & e^{-\frac{2}{\epsilon}} \\
< \; & 0.14,
\end{aligned}$$

so G is rejected with probability at least $0.86 > \frac{2}{3}$.

(2) Consider the case that the set U computed by the oracle is of size at most $\epsilon n/2$. With probability at most 1 the algorithm enters Stage II (Step 12). Note that G can be accepted by Algorithm Treewidth-Tester only when the algorithm enters Stage II. Then, by the definition of U we derive that every connected component of $G - U$ is of size at most $\zeta(k, d, \epsilon)$, and hence of diameter no more than $\zeta(k, d, \epsilon) - 1$. In other words, by removing at most $\epsilon dn/2$ edges that are incident with vertices in U from G we obtain a graph G' such that the diameter of every connected component of G' is no greater than $\zeta(k, d, \epsilon)$. By the assumption that G is ϵ-far from $tw(G) \leq k$, we know that G' is still $(\epsilon/2)$-far from $tw(G) \leq k$. This implies that at least $\epsilon n/2$ vertices belong to components of treewidth greater than k. Therefore, the probability that no vertex of S selected in Step 13 is in a connected component of treewidth greater than k is $(1-\epsilon/2)^{t_2}$, which is at most $((1-\epsilon/2)^{-2/\epsilon})^{-2} < e^{-2} < 0.14$ by setting $t_2 = 4/\epsilon$. Thus in this case, G is rejected with probability at least 0.86.

By the above analysis in cases (1) and (2), the probability that G is accepted is

$$\begin{aligned}
& \mathbf{Pr}\left[G \text{ is accepted by Algorithm } \textsf{Treewidth-Tester} : |U| > \frac{\epsilon n}{2}\right] + \\
& \mathbf{Pr}\left[G \text{ is accepted by Algorithm } \textsf{Treewidth-Tester} : |U| \leq \frac{\epsilon n}{2}\right] \\
\leq\ & 1 \cdot 0.14 + 1 \cdot 0.14 \\
\leq\ & \frac{1}{3}.
\end{aligned}$$

Therefore, the algorithm rejects G with probability at least $1 - 1/3 > 2/3$. The theorem is then proved. □

By Lemma 5.4, we know that the number of vertices in U found in Algorithm Global-Partition is at most $\epsilon n/4$ with probability $1 - 8d/\beta = 82/90$ by taking $\beta = 90d$. Next, we describe how to simulate Algorithm Global-Partition in time independent of n to have an efficient constant-time partitioning oracle.

5.2.4 Simulating the partitioning oracle in constant-time

Given a vertex v of $G = (V, E)$, the partitioning oracle has to answer the question that whether v is in U or v is in $(\mathcal{P}[u])_I$ for some $u \in V$. To fulfill this task efficiently, in the following we slightly modify the algorithm proposed by Hassidim et al. [83] which simulates the oracle locally.

The simulating algorithm. Instead of generating a random permutation in Algorithm Global-Partition, for each query of a vertex $v \in V$, we independently assign v a number $r(v)$ in $[0, 1]$ uniformly at random[1], and then compute I_v and S_v. Note that we only generate $r(v)$ when it is necessary. To compute I_v and S_v, we first recursively compute I_u and S_u for each u with $r(u) < r(v)$ and distance to v at most λ, where λ will be determined later. If $v \in I_u$ for one of those u, then set $I_v = I_u$, $S_v = S_u$, and return $v \notin U$. If $v \in S_u$ for one of those u, then return $v \in U$. Otherwise, we exhaustively search for a nonexpanding component containing v. If such a nonexpanding component, say I, is found, we set $I_z = I$ and $S_z = N_G(I)$ for all $z \in I$. If no such a nonexpanding component exists, we set $I_v = \{v\}$ and

[1] An arbitrary random real numbers in $[0, 1]$ cannot be generated in practice, nevertheless, it suffices to "discretize" the range $[0, 1]$ so that the probability that two edges are assigned the same number is negligibly small.

$S_v = N_G(v)$ and return $v \notin U$. Note that all vertices in I_u that we recursively computed are no longer left in the graph.

We set $\lambda = 2 \cdot \zeta(k, d, \epsilon) \le 8\beta^2 d(k+1)/\epsilon^2 = 64800 d^3(k+1)/\epsilon^2$ due to the following reasons. For each vertex u with $r(u) < r(v)$, I_u and S_u are supposed to be computed earlier than I_v and S_v. Moreover, since the graph induced by a nonexpanding set I and its open neighborhood $N_G(I)$ has diameter at most $\zeta(k, d, \epsilon)$, a vertex u could be either in $I_v \cup S_v$ or in $I_w \cup S_w$, where v and w are two vertices of distance $\lambda = 2 \cdot \zeta(k, d, \epsilon)$ in the graph. Thus, I_v and S_v should be computed after I_u and S_u, for all vertices u with $r(u) < r(v)$ and distance to v at most λ, are computed.

Let $\hat{G} = (V, \hat{E})$ be a graph where $\hat{E} = \{(u, v) \mid u, v \in V, d_G(u, v) \le \lambda\}$. The degree of $\hat{G}$ is bounded by $d + d^2 + \ldots + d^\lambda = d(d^\lambda - 1)/(d-1) < 2d^\lambda$. We denote that $D = 2d^\lambda$. We define a function $f_r : V \mapsto A$ recursively, using a function $g : V \times (V \times A)^* \mapsto A$, where $A = \bigcup_{v \in V} (\mathcal{P}[v])_I \cup U$, as follows. For each vertex v, we define that

$$f_r(v) = g(v, \{(u, f_r(u)) \mid (v, w) \in \hat{E},\ r(u) < r(v)\}).$$

The function value $f_r(v)$ depends on $f_r(u)$ for $r(u) < r(v)$ and $(v, u) \in \hat{E}$ (i.e., u is of distance at most D in the graph G). Clearly, $f_r(v)$ corresponds to a query to a partitioning oracle $\mathcal{O}$ at a vertex v. Note that the computation time required to compute $f_r(v)$ is in proportion to the number of queries incurred in its recursive computation. These queries form a rooted tree when we regard each of them as a node of the tree and $f_r(v)$ as the root. Hence the number of queries incurred during computing $f_r(v)$ is equal to the size of such a tree of recursion, which consists of paths starting at the node $f_r(v)$. Lemma 5.7, which is basically proved by Nguyen and Onak [97], gives the expected number of queries to $\mathcal{O}$ during computing $f_r(v)$. To make our analysis self-contained, we present the lemma as well as its proof as follows.

Lemma 5.7 (Nguyen and Onak [97], Lemma 12)**.** *Given $\hat{G} = (V, \hat{E})$ with $\Delta(\hat{G}) \le D$ and for each query to the partitioning oracle $\mathcal{O}$ at a vertex $v \in V$, the simulating algorithm computes $f_r(v) = g(v, \{(u, f_r(u)) \mid (v, w) \in \hat{E},\ r(u) < r(v)\})$, then the expected number of queries to $\mathcal{O}$ performed by the algorithm during computing $f_r(v)$ is at most 4^D.*

Proof. To compute $f_r(v)$, the simulating algorithm starts from computing $f_r(v)$ with $r(v) \in [0,1]$ chosen uniformly at random and explores all paths $w_0 = v, w_1, \ldots, w_h$ in $\hat{G}$ such that $r(w_0) > r(w_1) > \ldots > r(w_h)$ such that $r(w_0) > r(w_1) > \ldots > r(w_h)$ for some integer $h \geq 0$. Let $Q(x)$ be an upper bound on the expected number of queries to $\mathcal{O}$ for any vertex v with $r(v) = x$. By the definition of $f_r(v)$ we know that $Q(x) \leq Q(y)$ whenever $x \leq y$. Let $u_1, u_2, \ldots, u_\ell$ be the neighbors of v in $\hat{G}$, where $\ell \leq D$, and let $r(u_i) = y_i$ for each $i \in [\ell]$. To compute $f_r(v)$, we first examine its neighbors, and then for each of its neighbors u_i with $y_i < r$, we explore all paths starting from u_i. The expected number of queries incurred on each path starting from u_i is then bounded by $Q(y_i)$. Thus we have $Q(x) \leq 1 + \sum_{i=1}^{\ell} \mathbf{Pr}[y_i \leq x] \cdot \mathbf{E}[Q(y_i) \mid y_i < x]$. If we substitute x by $i/2D$ for $1 \leq i \leq 2D$, we have

$$\begin{aligned}
Q\left(\frac{i}{2D}\right) &\leq 1 + \sum_{j=1}^{D} \mathbf{E}\left[Q(y_j) \,\middle|\, y_j < \frac{i}{2D}\right] \cdot \mathbf{Pr}\left[y_j < \frac{i}{2D}\right] \\
&\leq 1 + \sum_{j=1}^{D} \sum_{h=1}^{i} \mathbf{E}\left[Q(y_j) \,\middle|\, y_j \in \left[\frac{h-1}{2D}, \frac{h}{2D}\right)\right] \cdot \mathbf{Pr}\left[y_j \in \left[\frac{h-1}{2D}, \frac{h}{2D}\right)\right] \\
&\leq 1 + \sum_{j=1}^{D} \sum_{h=1}^{i} Q\left(\frac{h}{2D}\right) \cdot \frac{1}{2D} \\
&\leq 1 + \frac{1}{2} \cdot \sum_{h=1}^{i} Q\left(\frac{h}{2D}\right) \\
&= 1 + \frac{1}{2} \cdot Q\left(\frac{i}{2D}\right) + \frac{1}{2} \cdot \sum_{h=1}^{i-1} Q\left(\frac{h}{2D}\right).
\end{aligned}$$

Hence we obtain that

$$Q\left(\frac{i}{2D}\right) \leq 2 + \sum_{h=1}^{i-1} Q\left(\frac{h}{2D}\right).$$

We prove that $Q(i/2D) \leq 2^i$ by induction on i as follows. For $i = 1$, it clearly holds that $Q(1/2D) = 2 \leq 2^1$. Assume that it holds for $i \leq a-1$, $a \geq 2$. Then for $i = a$,

$$Q\left(\frac{h}{2D}\right) \leq 2 + \sum_{h=1}^{a-1} Q\left(\frac{h}{2D}\right) \leq 2 + \sum_{h=1}^{a-1} 2^h = 2^a.$$

Therefore, since $Q(.)$ is a monotonically increasing, for each $v \in V$ we have

$$Q(r(v)) \leq Q(1) \leq 2^{2D} = 4^D.$$

Thus, the lemma is proved. $\square$

Lemma 5.7 implies that the expected number of queries to the oracle performed in Stage I of Algorithm Treewidth-Tester is at most $t_1 \cdot 4^D$. To have a bounded number of queries to the oracle, we modify Algorithm Treewidth-Tester by halting the oracle and ending the execution of Algorithm Treewidth-Tester (at Step 5) whenever a query to the oracle incurs more than $25600 \cdot 4^D/\epsilon^2$ queries. By Markov's inequality, such an event happens in a query with probability at most $4^D/(25600 \cdot 4^D/\epsilon^2) = \epsilon^2/25600$. By the union bound, the probability that this event happens during any one of the t_1 queries is at most $t_1 \cdot \epsilon^2/25600 = (256/\epsilon^2) \cdot \epsilon^2/25600 = 0.01$. Although the modified Algorithm Treewidth-Tester may stop executing with probability at most 0.01, together with the error probability clarified in the proof of Theorem 5.3, Theorem 5.3 still holds. The total number of queries to the oracle is then bounded by $O(t_1 \cdot 4^D/\epsilon^2) = O(4^D/\epsilon^4)$.

For each vertex v of G, a nonexpanding component containing v can be found in $O(2^{d^{\zeta(k,d,\epsilon)-1}} \cdot d \cdot \zeta(k,d,\epsilon))$ time by exhaustively exploring the subsets of vertices that are at distance at most $\zeta(k,d,\epsilon) - 1$ from v, and then checking if any one of these subsets fulfills the three conditions of Definition 5.1. The following lemma shows that the complexity of finding such a nonexpanding set can be improved to $O(2^{\zeta(k,d,\epsilon)\log\zeta(k,d,\epsilon)})$ by exhaustively examining *connected* induced subgraphs containing v.

Lemma 5.8. *Given a graph $G = (V, E)$ with $\Delta(G) = d$ and a designated vertex v. Then all the connected induced subgraphs of G of size at most α that contain v can be found in $O((\alpha - 1)!d^{\alpha-1})$ time. Furthermore, a nonexpanding component containing v can be found in $O(2^{\zeta(k,d,\epsilon)\log(d\cdot\zeta(k,d,\epsilon))})$ time.*

Proof. To search for a connected induced subgraph of G of size at most α that contains the designated vertex v, we use a search tree algorithm which works recursively and starts branching at v to exhaustively search for connected components containing v. Whenever the algorithm branches on a vertex u, it visits u then recursively branches on every unvisited vertex which is adjacent to any visited one. Whenever the number of visited vertices achieves α, the algorithm stops branching (the recursion stops). The behavior of this algorithm can be represented as a tree, say $\mathfrak{T}$. Each tree node of $\mathfrak{T}$ corresponds to a vertex on which the algorithm branches. Thus, the path on $\mathfrak{T}$ from the root to any node corresponds to a connected induced subgraph of G. In particular, the path on $\mathfrak{T}$ from the root to any leaf node corresponds to a connected induced subgraph of G with α vertices. Since it is clear that every connected induced subgraph of G containing v can be found by this search tree algorithm, the number of induced subgraphs of G of size at most α that contains v is bounded by the number of nodes of $\mathfrak{T}$. Next, we prove by induction on α that the number of nodes in $\mathfrak{T}$, say $T(\alpha)$, is at most $(\alpha-1)!d^{\alpha-1}$.

Since $T(1) = 1$ (i.e., the connected induced subgraph of G is simply a vertex v), it is easy to see that $T(\alpha) \leq (\alpha-1)!d^{\alpha-1}$ holds for $\alpha = 1$. Assume that $T(\alpha) \leq (\alpha-1)!d^{\alpha-1}$ holds for $\alpha = \ell - 1$, that is, $T(\ell-1) \leq (\ell-2)!d^{\ell-2}$. Since each vertex in a connected induced subgraph of G has at most $d-1$ neighbors that are unvisited by the algorithm and each connected induced subgraph of G computed for $T(\ell-1)$ has at most $\ell-1$ vertices, we obtain that

$$\begin{aligned}
T(\ell) &= T(\ell-1) + T(\ell-1)\cdot(\ell-1)(d-1) \\
&= (\ell-2)!d^{\ell-2} + (\ell-1)!d^{\ell-2}\cdot(d-1) \\
&= (\ell-2)!d^{\ell-2} + (\ell-1)!d^{\ell-1}\cdot(1-1/d) \\
&= (\ell-1)!d^{\ell-1} + ((\ell-2)!d^{\ell-2} - (\ell-1)!d^{\ell-2}) \\
&\leq (\ell-1)!d^{\ell-1}.
\end{aligned}$$

Thus, by the principle of mathematical induction, we proved that $T(\alpha) \leq (\alpha-1)!d^{\alpha-1}$. Furthermore, for each connected induced subgraph of G of size at most $\zeta(k,d,\epsilon)$ that contains v, we can check whether it is a nonexpanding component by examining its neighbors. Therefore, the time complexity for finding a nonexpanding component containing v is

$$\begin{aligned}
& O((\zeta(k,d,\epsilon)-1)!d^{\zeta(k,d,\epsilon)-1}\cdot d\cdot\zeta(k,d,\epsilon)) \\
= \ & O(\zeta(k,d,\epsilon)!d^{\zeta(k,d,\epsilon)}) \\
= \ & O(2^{\zeta(k,d,\epsilon)\log\zeta(k,d,\epsilon)}\cdot 2^{\log d^{\zeta(k,d,\epsilon)}}) \\
= \ & O(2^{\zeta(k,d,\epsilon)\log\zeta(k,d,\epsilon)}\cdot 2^{\zeta(k,d,\epsilon)\log d}) \\
= \ & O(2^{\zeta(k,d,\epsilon)\log(d\zeta(k,d,\epsilon))}).
\end{aligned}$$

$\square$

Therefore, the time complexity for computing all the queries to $\mathcal{O}$ is $O((4^D/\epsilon^4) \cdot 2^{\zeta(k,d,\epsilon)\log(d\cdot\zeta(k,d,\epsilon))}) = O(4^{2d^\lambda} \cdot 2^{\zeta(k,d,\epsilon)\log(d\zeta(k,d,\epsilon))}/\epsilon^4) = O(2^{4d^\lambda+\zeta(k,d,\epsilon)\log(d\zeta(k,d,\epsilon))}/\epsilon^4) = O(2^{4d^{2\zeta(k,d,\epsilon)}+\zeta(k,d,\epsilon)\log(d\zeta(k,d,\epsilon))}/\epsilon^4)$. Thus, we have the following lemma.

Lemma 5.9. *The time complexity of Stage I of Algorithm* Treewidth-Tester *is*

$$O\left(\frac{2^{4d^{2\zeta(k,d,\epsilon)}+\zeta(k,d,\epsilon)\log(d\zeta(k,d,\epsilon))}}{\epsilon^4}\right),$$

where $\zeta(k,d,\epsilon) = 32400d^3(k+1)/\epsilon^3$.

Theorem 5.4. *The time complexity of Algorithm* Treewidth-Tester *is*

$$O\left(\frac{2^{4d^{2\zeta(k,d,\epsilon)}+\zeta(k,d,\epsilon)\log(d\zeta(k,d,\epsilon))}}{\epsilon^4} + \frac{c^{k^3} \cdot d^{\zeta(k,d,\epsilon)-1}}{\epsilon}\right) = 2^{d^{O(kd^3/\epsilon^2)}},$$

where $\zeta(k,d,\epsilon) = 32400d^3(k+1)/\epsilon^2$, *and* $c > 1$ *is a constant.*

Proof. In Stage II, we select $t_2 = 4/\epsilon$ vertices in G, each of them is contained in some component of size at most $d^{\zeta(k,d,\epsilon)-1}$. Since checking whether one of the above connected components has treewidth at most k can be done in $O(c^{k^3} \cdot d^{\zeta(k,d,\epsilon)-1})$ time for a constant $c > 1$ [28], to see whether the induced subgraph of these t_2 vertices has treewidth at most k only takes $O(t_2 \cdot c^{k^3} \cdot d^{\zeta(k,d,\epsilon)-1}) = O(c^{k^3} \cdot d^{\zeta(k,d,\epsilon)-1}/\epsilon)$ time. Thus, together with Lemma 5.9 we obtain that the running time complexity of Algorithm Treewidth-Tester as claimed in the theorem. □

5.2.5 An improved partitioning oracle

Czygrinow et al. [55] proposed distributed approximation algorithms for several **NP**-hard optimization problems. Inspired by the partitioning algorithm used in [55], Onak [102] proposed a distributed algorithm to derive a much simpler partitioning oracle for minor-closed properties. Using this simple and efficient partitioning oracle, Onak derived an $O(d^{\mathrm{poly}(1/\epsilon)})$ property tester for minor-closed properties. we show that Algorithm Treewidth-Tester runs in time $d^{(k/d)^{O(k^2)}} + 2^{\mathrm{poly}(k,d,1/\epsilon)}$ based on the approaches in [55, 102] for constructing an efficient partitioning oracle.

The *arboricity* of an undirected graph $G = (V, E)$ is the minimum number of forests into which E can be partitioned. Compared with the facts used in [102] on H-minor free graphs for an arbitrary fixed minor H, we use the following facts about treewidth and arboricity of a graph. Fact 5.1 is mentioned in Bodlaender's work in [28], and Fact 5.2 follows from the well-known Nash-Williams Theorem [91] and Proposition 2 in [61].

Fact 5.1 ([28]). For every finite graph $G = (V, E) \in \mathcal{P}_{tw \leq k}$, $|E| < k \cdot |V|$.

Fact 5.2 ([61, 91]). For every finite graph $G = (V, E) \in \mathcal{P}_{tw \leq k}$, E can be partitioned into at most k forests.

Assume that the input graph has weights on its edges. The following we define what a *partition contraction* is.

Definition 5.5. Let $(V_1, \ldots, V_p)$ be a partition of the vertex set V of a weighted graph $G = (V, E, w)$, where $p \geq 1$ is a positive integer. The *partition contraction* $G \mid (V_1, \ldots, V_p)$ of G with respect to $(V_1, \ldots, V_p)$ is a weighted graph $G' = (V', E', w')$ such that the following conditions hold.

- $V' = \{z_1, \ldots, z_p\}$, where z_i corresponds to V_i for $i \in [p]$;
- $E' = \{(z_i, z_j) \mid z_i, z_j \in V',$ and there exist $v_i \in V_i, v_j \in V_j,\ i \neq j,$ such that $(v_i, v_j) \in E\}$;
- $w'((z_i, z_j)) = \sum_{v \in V_i,\, v' \in V_j} w(v, v')$.

Note that if $G[V_i]$ is connected for each $i \in [p]$, then $G \mid (V_1, \ldots, V_p)$ is actually formed by a series of edge contractions in $G[V_i]$'s. Hence the following fact holds since $\mathcal{P}_{tw \leq k}$ is minor-closed.

Fact 5.3. Let $G = (V, E)$ be a graph in $\mathcal{P}_{tw \leq k}$ and $(V_1, \ldots, V_p)$ be a partition of V. If $G[V_i]$ is connected for each $i \in [p]$, then $G \mid (V_1, \ldots, V_p)$ is also in $\mathcal{P}_{tw \leq k}$.

Algorithm Improved-Partition iterates for $7 \cdot (36k - 1) \cdot \lceil \log_{(1-1/(36k))}(\epsilon/k) \rceil$ times. Initially, each edge of the input graph G has weight 1. In each iteration (i.e., Line 5–25), the algorithm finds stars (i.e., a tree with exactly one internal node and other vertices as leaves). For each of these stars, each leaf, say z_j, is labelled by 1, the internal node, say z_i, is labelled by 0, and the edge weight of (z_i, z_j) is maximum among all the edges incident to z_j. Then, the algorithm contracts these stars (see Lines 16–25) to construct the corresponding partition contraction and proceeds to

Improved-Partition(G)
/* G: a weighted graph stored in an adjacency list with $\Delta(G) \le d$. */
/* Each edge of G has weight 1 initially. */
begin
$\mathcal{P} \leftarrow \{\{v\} \mid v \in V\}$; /* the initial partition of V; */
/* $\tilde{G} \leftarrow G \mid (\{v_1\}, \{v_2\}, \ldots, \{v_n\})$ for $v_i \in V$ */;
$p \leftarrow n$;
$V_i \leftarrow \{v_i\}$ and z_i stands for V_i for each $i \in [p]$;
repeat
  for each vertex z_i of $\tilde{G}$ **do**
    label z_i by a number in $\{0, 1, 2\}$ uniformly at random;
  end for
  for each vertex z_i of $\tilde{G}$ **do**
    $z'_i \leftarrow \arg\max_{z_j \in N_{\tilde{G}}(z_i)} w(z_i, z_j)$;
  end for
  for each vertex z_i of $\tilde{G}$ **do**
    if z_i is labelled by 0 **then**
      construct $\mathcal{L}_i = \{j \in [p] \mid z'_j = z_i$ and z_j is labelled by $1\}$;
    end if
  end for
  for each vertex z_i of $\tilde{G}$ **do** /* contract the 0-1 stars and update $\tilde{G}$ */
    if z_i is labelled by 0 **then**
      $V_i \leftarrow V_i \cup \bigcup_{j \in \mathcal{L}_i} V_j$;
      $E(\tilde{G}) \leftarrow E(\tilde{G}) \cup \left\{(z_i, z_h) \mid z_h \in N_{\tilde{G}}\left(\bigcup_{j \in \mathcal{L}_i} z_j\right) \setminus N_{\tilde{G}}[z_i]\right\}$;
      for $z_h \in N_{\tilde{G}}\left(\bigcup_{j \in \mathcal{L}_i} z_j\right) \setminus \{z_i\}$ **do**
        $w(z_i, z_h) \leftarrow w(z_i, z_h) + \sum_{j \in \mathcal{L}_i, u \in V_j, v \in V_h} w(u, v)$;
      end for
      remove $\bigcup_{j \in \mathcal{L}_i} z_j$ from $\tilde{G}$;
    end if
  end for
until $7 \cdot (36k - 1) \cdot \lceil \log_{(1-1/(36k))}(\epsilon/k) \rceil$ times
end

Algorithm 5.5: Improved-Partition: an improved partitioning oracle for $\mathcal{P}_{tw \le k}$.

next iteration. Note that these stars found in each iteration, say them *0-1 stars*, are vertex-disjoint due to the following reasons. Every vertex z_i is labelled by exactly one number in $\{0, 1, 2\}$ (by Line 6). Suppose that a vertex z_i is labelled by 1, then it can only be selected as a leaf of a star by the algorithm. Since the algorithm picks exactly one neighbor z'_i of z_i, z_i can only be in at most one found star. On the other hand (i.e., z_i is labelled by 0), z_i can only be selected as an internal node of a star by the algorithm, and hence it can only be in at most one found star.

As the 0-1 stars found in each iteration are vertex-disjoint, to simplify the analysis of the time complexity of the algorithm, we regard Algorithm Improved-Partition as a *local distributed algorithm*. It runs on a synchronous network G, where each vertex corresponds to a computation unit and each edge represents an underlying communication link. Such an algorithm consists of a constant number of rounds. In each communication round, every vertex in G can send messages to all its neighbors, receive messages from all its neighbors, and perform some local computations. Note that in each iteration of Lines 5–25, the number of communication rounds of the algorithm is $O(1)$.

Lemma 5.10. *Let* $(V_1, \ldots, V_p)$ *be a partition of* V *of a graph* $G = (V, E, w) \in \mathcal{P}_{tw \leq k}$ *such that* $G[V_i]$ *is connected for each* $i \in [p]$*. Then Line 5–25 in Algorithm* Improved-Partition *turns* $G_1 = G \mid (V_1, \ldots, V_p)$ *into a graph* $G_2 = G \mid (V'_1, \ldots, V'_{p'})$ *such that with probability at least* $1/(36k - 1)$ *the total weight of edges in* G' *is at most* $(1 - 1/(36k))$ *of the total weight of edges in* G*.*

Proof. Let W be the sum of all the edge weights in G_1. By Fact 5.2, we know that the edge set of G_1 can be partitioned into at most k forests. Thus, at least one of these forests has weight at least W/k by the pigeonhole principle. Note that if we root every tree in this forest and put orientation on each edge towards the corresponding root, then there is at most one edge directed from each vertex in the forest. We denote by a_v the weight of such an edge directed from vertex v. When running Algorithm Improved-Partition on G_1, every vertex z_i (regarded as a computation unit) selects an incident edge with maximum weight (Line 9), which is clearly at least a_{z_i}, and an edge can be selected at most twice by its two endpoints, the total weight of selected edges is at least $W/2k$. By the labelling process (Line 6), we have that each of these edges is contracted with probability 1/9 (i.e., the probability that z_i is labelled by 1 and z_i' is labelled by 0 for a selected edge (w_i, w_i')). Thus the expected weight of contracted edges is $(W/2k)/9 = W/18k$. By Markov's inequality, we obtain that with probability at most

$$\frac{1-1/(18k)}{1-1/(36k)} = 1 - \frac{1}{36k-1}$$

the total weight of uncontracted edges is greater than $W(1-1/(36k))$. Hence, the proposition is proved. $\square$

Proposition 5.3. *Let $\epsilon \in (0,1)$. For every graph $G=(V,E) \in \mathcal{P}_{tw\le k}$, there exists a local distributed partitioning algorithm that requires $(k/\epsilon)^{O(k^2)}$ communication rounds and determines a partition $(V_1,\dots,V_p)$ of G such that:*

- *the diameter of each connected component V_i is $(k/\epsilon)^{O(k^2)}$;*
- *the number of cut edges is at most $\epsilon|V|$ with probability at least 82/90.*

Furthermore, the total amount of computation for each vertex is bounded by $d^{(k/\epsilon)^{O(k^2)}}$.

***Proof*.** Algorithm Improved-Partition iteratively runs the loop in Lines 5–25 for $7 \cdot (36k-1) \cdot \lceil \log_{(1-1/(36k))}(\epsilon/k) \rceil$ times. Each iteration produces a partition of V based on the one obtained in the previous iteration. Here we claim that the diameter of each connected component in the ith iteration is bounded by $3^i - 1$. We prove the claim by induction on i as follows. For $i=0$, each connected component is simply a single vertex hence the claim is clearly true. Assume that the claim holds for $i \le \ell - 1$ and denote by d_i the diameter of each connected component in the ith iteration. Recall that each connected component in the ℓth iteration is formed by contracting a 0-1 star in which each node of the star corresponds to a connected component formed in the $(\ell-1)$th iteration. It is easy to see that the diameter of each connected component in the ℓth iteration is bounded by $3d_{\ell-1}+2$ due to the structure of a star. Since $d_{\ell-1} \le 3^{\ell-1}-1$ by induction hypothesis, we have $d_\ell \le 3d_{\ell-1}+2 \le 3(3^{\ell-1}-1)+2 \le 3^\ell - 1$. Hence the claim is proved. Thus we obtain that at the end of all the iterations, the diameter of each connected component is bounded by $3^{7\cdot(36k-1)\cdot\lceil \log_{(1-1/(36k))}(\epsilon/k)\rceil} \le 3^{7\cdot 36k \cdot \log_{(1+1/(36k-1))}(k/\epsilon)} \cdot 3^{7\cdot 36k}$. Since

$$\log_{1+\frac{1}{36k-1}}\left(\frac{k}{\epsilon}\right) \le \log_{1+\frac{1}{36k}}\left(\frac{k}{\epsilon}\right) = \log_3\left(\frac{k}{\epsilon}\right)^{\log_3^{-1}(1+1/(36k))},$$

we have

$$\begin{aligned}
3^{7\cdot 36k\cdot \log_{(1+1/(36k-1))}(k/\epsilon)} &= \left(\frac{k}{\epsilon}\right)^{\frac{7\cdot 36k}{\log_3(1+1/(36k))}} \\
&\le \left(\frac{k}{\epsilon}\right)^{\frac{7\cdot 36k}{\log_{e^2}(1+1/(36k))}} \\
&\le \left(\frac{k}{\epsilon}\right)^{\frac{2\cdot 7\cdot 36k}{1/(36k)-1/(36k)^2/2}} \\
&= \left(\frac{k}{\epsilon}\right)^{O(k^2)}.
\end{aligned}$$

Thus, the diameter of each connected component is eventually bounded by $(k/\epsilon)^{O(k^2)}$. The number of required communication rounds of the algorithm is then bounded by $(k/\epsilon)^{O(k^2)}$.

By Lemma 5.10, an iteration of the loop decreases the number of edges cut by the current partition by a factor of at most $1 - 1/(36k)$ with probability at least $1/(36k-1)$. Thus, the expected number of times that this happens is at least $7\cdot\lceil\log_{(1-1/(36k))}(\epsilon/k)\rceil$. By the Chernoff bound, we have that this happens fewer than $\lceil\log_{(1-1/(36k))}(\epsilon/k)\rceil$ times with probability at most $e^{-7\cdot(6/7)^2/2} < 8/90$. Therefore, the algorithm finally produces a partition that cuts at most

$$|E|\cdot(1-1/(36k))^{\log_{(1-1/(36k))}(\epsilon/k)} < k|V|\cdot(\epsilon/k) = \epsilon|V|$$

edges with probability at least 82/90. Since the size of each resulting connected component is bounded by $d^{(k/\epsilon)^{O(k^2)}}$ and the degree of each vertex in any partition contraction of G is bounded by $d\cdot d^{(k/\epsilon)^{O(k^2)}} = d^{(k/\epsilon)^{O(k^2)}}$, the total amount of computation for each vertex is bounded by $d^{(k/\epsilon)^{O(k^2)}}$. □

In order to derive an $(\epsilon/4, \zeta(k,d,\epsilon))$-partitioning oracle for $\mathcal{P}_{tw\le k}$, we consider the following further construction work. We substitute ϵ in Algorithm Improved-Partition by $\epsilon/8$. For each cut edge, we distinguish arbitrary one of its endpoints by a cut vertex and add it into U. To simulate Algorithm Improved-Partition on a queried vertex v, we first generate the subgraph $B_G(v,2r)$ of G induced by vertices with distance $2r$ from v, where $r = 2^{7\cdot(36k-1)\cdot\lceil\log_{(1-1/(36k))}(\epsilon/k)\rceil} = (k/\epsilon)^{O(k^2)}$. As the algorithm simulating a local distributed algorithm [103], we run Algorithm Improved-Partition sequentially on $B_G(v,2r)$, which requires additional factor $|B_G(v,2r)| \le$

$d^{(k/d)^{O(k^2)}}$ of the running time. Note that it makes the same decision about v as it runs for r rounds on the whole graph G. No information originated from a vertex with distance greater than r from v can reach v. The additional factor 2 of the term $B_G(v, 2r)$ is due to that reason that the decision on the vertices at distance exactly r from v can depend on those at distance at most $2r$ from v. Next, let us consider the following well-known theorem.

Theorem 5.5 ([27, 30, 105])**.** *Let $G = (V, E) \in \mathcal{P}_{tw \leq k}$, then there exists a subset $S \subseteq V$ of size at most k such that removing S from G results in connected components of size at most $|V|/2$. Moreover, there exists an $O(k^2 \cdot |V|)$ algorithm to find such a set S.*

Assume that C is a connected component obtained by Improved-Partition for the queried vertex v and $|C|$ is larger than $\zeta(k, d, \epsilon)$. Using Theorem 5.5, we further recursively partition C into smaller connected components until each component is of size at most $\zeta(k, d, \epsilon)$. Note that the computation of these smaller connected components is independent of which vertex is the queried vertex. We add the vertices removed during this further recursive partitioning algorithm into U. Since $\zeta(k, d, \epsilon) = O(32400 d^3 (k + 1)/\epsilon^2)$, the depth of the recursion is at most $\log(d^{(k/\epsilon)^{O(k^2)}}/(kd^3/\epsilon^2)) \leq (\log d) \cdot (k/\epsilon)^{O(k^2)}$. During each recursion of the further partitioning algorithm, the ratio of the number of removed vertices to the size of each connected component is at most $k/(kd^3/\epsilon^2) = \epsilon^2/d^3 \leq \epsilon/8$. Thus, plus the previous $\epsilon n/8$ cut vertices added by Algorithm Improved-Partition, we derive that $|U| \leq \epsilon n/4$. Hence, we obtain an $(\epsilon/4, \zeta(k, d, \epsilon))$-partitioning oracle for $\mathcal{P}_{tw \leq k}$.

Using the recursion-tree method [51], we obtain that the total time complexity of this further recursive partitioning algorithm is $(k^2 \cdot d^{(k/\epsilon)^{O(k^2)}}) \cdot (\log d) \cdot (k/\epsilon)^{O(k^2)} = d^{(k/\epsilon)^{O(k^2)}}$. Therefore, substitute the partitioning oracle in Sect. 5.2.4 by the above one for $\mathcal{P}_{tw \leq k}$, we obtain the following theorem.

Theorem 5.6. *The time complexity of Algorithm* Treewidth-Tester *is*

$$O\left(d^{(k/\epsilon)^{O(k^2)}} + \frac{c^{k^3} \cdot d^{\zeta(k,d,\epsilon)-1}}{\epsilon}\right) = d^{(k/\epsilon)^{O(k^2)}} + 2^{\mathrm{poly}(k,d,1/\epsilon)},$$

where $\zeta(k, d, \epsilon) = 32400 d^3 (k + 1)/\epsilon^2$, and $c > 1$ is a constant.

Chapter 6

Concluding Remarks and Future Work

6.1 Minimum Quartet Inconsistency and Minimum Triplet Inconsistency

There are another aspect of evolutionary tree reconstruction which focuses on *rooted* evolutionary trees [1, 36, 73, 81, 96, 115]. A rooted evolutionary tree T is a rooted, leaf-labeled binary tree such that the leaves of T are bijectively labeled by the taxa in the taxon set S, and each internal node of T has exactly two children (see Figure 6.1 for an illustrating example).

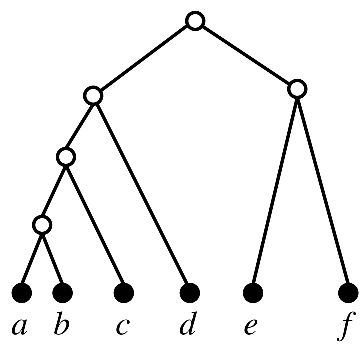

Figure 6.1: A rooted evolutionary tree with six leaves.

Similar to quartets and quartet topologies, triplets and triplet topologies can be defined as follows. A *triplet topology* is an evolutionary tree with three leaves. A triplet $\{a, b, c\}$ has a triplet topology either $((ab)c)$, $((ac)b)$, or $((bc)a)$ induced by a rooted evolutionary tree, as Fig. 6.2 shows. Let Y be a set of triplet topologies over S. We say that Y is *complete* if each triplet over S has exactly one topology in Y. We denote by Y_T the set of all induced triplet topologies in a rooted evolutionary tree T.

We say that Y is *rooted tree-consistent* if there exists a rooted evolutionary tree T such that $Y \subseteq Y_T$. The *parameterized Minimum Triplet Inconsistency problem* (parameterized MTI) is defined as follows.

The Parameterized Minimum Triplet Inconsistency problem (parameterized MTI)
Input: Given a complete set of $\binom{n}{3}$ triplet topologies Y over a set S of n taxa and a parameter k
Task: Determine if changing at most k triplet topologies in Y makes Y rooted tree-consistent.

For the case that the set Y of triplet topologies is not necessarily complete, determining if a set of triplet topologies Y is rooted tree-consistent can be done in polynomial time [1] (In particular, it is $O(\min\{|Y|n^{1/2}, |Y| + n^2 \log n\})$ solvable by Henzinger et al. [84]). This is different from that Quartet Compatibility problem for unrooted evolutionary trees. The *Maximum Consensus Tree from Rooted Triplets* problem (MCTT) is to find a rooted evolutionary tree that satisfies as many triplet topologies in Y as possible. Wu [115] proved that the MCTT problem is **NP**-hard. They also provided an $O((|Y| + n^2)3^n)$ algorithm for this problem.

For the case that the set Y of triplets topologies is complete (cf., *minimally dense* in [39, 81]), Byrka et al. [39] showed that the minimum Triplet Inconsistency problem is **NP**-hard, which implies that the parameterized MTI problem is **NP**-complete. Recently, Guillemot and Mnich [81] gave a subexponential fixed-parameter algorithm for the parameterized MTI problem, which runs in $2^{O(k^{1/3} \log k)} + O(n^4)$ time.

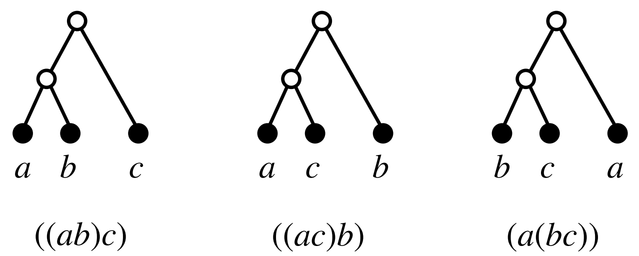

Figure 6.2: Possible topologies of a triplet $\{a, b, c\}$.

One might be curious about whether our approaches for solving the parameterized MQI problem can be applied to the parameterized MTI problem. Actually, Proposition 1 in [81] implies that Y is rooted tree-consistent if and only if every

$Y' \subseteq Y$ over four taxa is rooted tree-consistent. A set of four taxa $\{a, b, c, d\}$ is called a *local triplet conflict* if the set of triplet topologies over $\{a, b, c, d\}$ is not rooted tree-consistent. Due to this proposition, our approaches using depth-bounded search tree can be applied and it is likely to derive an $O^*((1 + \varepsilon)^k)$ fixed-parameter algorithm for this problem, where $\varepsilon > 0$ can be arbitrarily small. It is also interesting to devise a subexponential fixed-parameter algorithm for the parameterized MQI problem. It deserves to be noted that the subexponential fixed-parameter algorithm in [81] relies on an observation that one can focus on an obstruction subset $Y' \subseteq Y$ which involves the taxa belonging to local triplet conflicts. This observation is helpful since if Y is a "yes" instance of the parameterized MTI problem, then Y' involves at most $O(k^2)$ taxa. However, for the parameterized MQI problem, we could not obtain a similar obstruction $Q' \subseteq Q$ of size bounded by a function of k since every taxon appears in Q' when the input Q of quartet topologies is not tree-like.

In addition, similar to the property testing and parameterized property testing results on tree-consistency of quartet topologies, it is also interesting to consider testing rooted tree-consistency of triplet topologies.

6.2 Concluding Remarks and Future Work on Parameterized Property Testing

In Chapter 5, we have presented parameterized property testers for two graph properties $\mathcal{P}_{VC\leq k}$ and $\mathcal{P}_{tw\leq k}$ in the sparse model, both of which are weakly uniform on k. The parameterized problems corresponding to $\mathcal{P}_{VC\leq k}$ and $\mathcal{P}_{tw\leq k}$ both admit efficient fixed-parameter algorithms. This suggests the possibilities of devising parameterized property testers for graph properties whose corresponding parameterized problems are in **FPT**. Here, we propose a conjecture below.

Conjecture 6.1. *Every parameterized graph problem in* **FPT** *admits an* $O(\phi(k, 1/\epsilon))$ *parameterized property tester that is weakly uniform on* k*, where* ϕ *is an arbitrary function solely depending on the parameter* k *and* ϵ*.*

As clarified in Chapter 5, there are some graph properties that are trivial to test, even though their corresponding parameterized problems are in **FPT**. However, there exist graph theoretical problems that are hard in both respects. Take k-

coloring as an example. To determine if a graph admits a k-coloring is not in **FPT** since it is **NP**-complete for even for $k = 3$ [72]. On the other hand, testing k-colorability in the sparse model requires $\Omega(n)$ time [33]. This illustrates the cases where introducing parameters for the property testing is not much helpful.

Naturally, we could relax the constraint on the time complexity of a parameterized property tester to $\phi(k, 1/\epsilon)\cdot\text{poly}(n)$ for the properties corresponding to **NP**-hard problems, where ϕ is an arbitrary function that solely depends on k and ϵ. However, it might not be easy to devise such algorithms due to the reason that there are properties which do not admit $O(\text{poly}(n/\epsilon))$ property testers unless $\mathbf{NP} \subseteq \mathbf{BPP}$ [75].

Bibliography

[1] A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman: Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM J. Comput.* **10** (1981) 405–421.

[2] N. Alon: Testing subgraphs in large graphs. *Random Structures Algorithms* **21** (2002) 359–370.

[3] N. Alon, E. Fischer, M. Krivelevich, and M. Szegedy: Efficient testing of large graphs. *Combinatorica* **20** (2000) 451–476.

[4] N. Alon and M. Krivelevich: Testing k-colorability. *SIAM J. Discrete Math.* **15** (2002) 211–227.

[5] N. Alon, M. Krivelevich, I. Newman, and M. Szegedy: Regular languages are testable with a constant number of queries. *SIAM J. Comput.* **30** (2001) 1842–1862.

[6] N. Alon and A. Shapira: Testing subgraphs in directed graphs. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing* (STOC 2003), San Diego, California, pp. 700-709. Also: *J. Comput. System Sci.* **69** (2004) 354–382.

[7] N. Alon and A. Shapira: Testing satisfiability. *J. Algorithms* **47** (2003) 87–103.

[8] N. Alon and A. Shapira: Homomorphisms in graph property testing – A survey. *Electronic Colloquium on Computational Complexity* (ECCC 2005). Report No. 85.

[9] N. Alon and A. Shapira: A characterization of easily testable induced subgraphs. *Combin. Probab. Comput.* **15** (2006) 791–805.

[10] N. Alon and A. Shapira: Every monotone graph property is testable. *SIAM J. Comput.* **38** (2008) 505–522.

[11] N. Alon and A. Shapira: A characterization of the (natural) graph properties testable with one-sided error. *SIAM J. Comput.* **37** (2008) 1703–1727.

[12] N. Alon, A. Shapira, and B. Sudakov: Additive approximation for edge-deletion problems. *Anns.of Math.* **170** (2009) 371–411.

[13] S. Arnborg, D. G. Corneil, and A. Proskurowski: Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth.* **8** (1987) 277–284.

[14] S. Arnborg and A. Proskurowski: Linear time algorithms for **NP**-hard problems restricted to partial k-trees. *Discret. Appl. Math.* **23** (1989) 11–24.

[15] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy: Proof verification and the hardness of approximation problems. *J. ACM* **45** (1998) 501–555.

[16] R. Balasubramanian, M. Fellows, V. Raman: An improved fixed parameter algorithm for Vertex cover. *Inform. Process. Lett.* **65** (1998) 163–168.

[17] H.-J. Bandelt and A. Dress: Reconstructing the shape of a tree from observed dissimilarity data. *Adv. in Appl. Math.* **7** (1986) 309–343.

[18] T. Batu, L. Fortnow, R. Rubinfeld, W. D. Smith, and P. White: Testing that distributions are close. In *Proceedings of the 41st Symposium on Foundations of Computer Science* (FOCS 2000), 2000, pp. 259–269.

[19] A. Becker, R. Bar-Yehuda, and D. Geiger: Randomized algorithms for the loop cutset problem. *J. Artificial Intelligence Res.* **12** (2000) 219–234.

[20] A. Ben-Dor, B. Chor, D. Graur, R. Ophir, and D. Pelleg: From four-taxon trees to phylogenies: the case of mammalian evolution. In *Proceedings of the 2nd Annual International Conference on Research in Computational Molecular Biology* (RECOMB 1998), pp. 9–19.

[21] I. Benjamini, O. Schramm, and A. Shapira: Every minor-closed property of sparse graphs is testable. In *Proceedings of the 40th Annual ACM Symposium*

on Theory of Computing (STOC 2008), pp. 393–402. See also *Adv. Math.* **223** (2010) 2200–2218.

[22] C. Berge: *Graphs and hypergraphs.* North-Holland Mathematical Library, Vol. 6, North-Holland Publishing Co., Amsterdam, 1973.

[23] V. Berry and O. Gascuel: Inferring evolutionary trees with strong combinatorial evidence. *Theoret. Comput. Sci.* **240** (2000) 271–298.

[24] V. Berry, T. Jiang, P. E. Kearney, M. Li, and H. T. Wareham: Quartet cleaning: Improved algorithms and simulations. In *Proceedings of the 7th Annual European Symposium on Algorithms* (ESA 1999), Lecture Notes in Comput. Sci., Vol. 1643, Springer-Verlag, 1999, pp. 313–324.

[25] M. Blum, M. Luby, and R. Rubinfeld: Self-testing/correcting with applications to numerical problems. *J. Comput. System Sci.* **47** (1993) 549–595.

[26] H. L. Bodlaender: A tourist guide through treewidth. *Acta Cybernet.* **11** (1993) 1–22.

[27] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks: Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *J. Algorithms* **18** (1995) 238–255.

[28] H. L. Bodlaender: A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.* **25** (1996) 1305–1317.

[29] H. L. Bodlaender: Treewidth: Algorithmic techniques and results. In *Proceedings of the 22nd International Symposium on the Mathematical Foundations of Computer Science* (MFCS 1997), Lecture Notes in Comput. Sci., Vol. 1295, 1997, Springer-Verlag, pp. 19–36.

[30] H. L. Bodlaender and D. M. Thilikos: Treewidth for graphs with small chordality. *Discret. Appl. Math.* **79** (1997) 45–61.

[31] H. L. Bodlaender: A partial k-arboretum of graphs with bounded treewidth. *Theoret. Comput. Sci.* **209** (1998) 1–45.

[32] H. L. Bodlaender: Discovering treewidth. In *Proceedings of the 31st International Conference on Current Trends in Theory and Practice of Computer Science* (SOFSEM 2005), Lecture Notes in Comput. Sci., Vol. 3381, 2005, Springer-Verlag, pp. 1–16.

[33] A. Bogdanov, K. Obata, and L. Trevisan: A lower bound for testing 3-colorability in bounded-degree graphs. In *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science* (FOCS 2002), pp. 93–102.

[34] J. A. Bondy and U. S. R. Murty: *Graph Theory*. Graduate Texts in Mathematics, Vol. 244, Springer, New York, 2008.

[35] R. L. Brooks: On Coloring the Nodes of a Network. In *Mathematical Proceedings of the Cambridge Philosophical Society, Cambridge University Press*, Vol. **37**, 1941, pp. 194–197.

[36] D. Bryant: *Building Trees, Hunting for Trees, and Comparing Trees: Theory and Methods in Phylogenetic Analysis*. Ph.D. thesis, University of Canterbury, Christchurch, New Zealand, 1997.

[37] P. Buneman: The recovery of trees from measures of dissimilarity. In F. R. Hodson, D. G. Kendall, and P. Tautu, eds., Mathematics in the Archaeological and Historical Sciences, pp. 387–395. Edinburgh University Press, Edinburgh, 1971.

[38] J. Buss, J. Goldsmith: Nondeterminism within **P**. *SIAM J.Comput.* **22** (1993) 560–572.

[39] J. Byrka, S. Guillemot, and J. Jansson: New results on optimizing rooted triplets consistency. *Discrete Appl. Math.* **158** (2010) 1136–1147.

[40] M. Bender and D. Ron: Testing properties of directed graphs: acyclicity and connectivity. *Random Structures Algorithms* **20** (2002) 184–205.

[41] D. Bryant and M. Steel: Constructing optimal trees from quartets. *J. Algorithms* **38** (2001) 237–259.

[42] M.-S. Chang, L.-J. Hung, A. Langer, C.-C. Lin, and P. Rossmanith: Parameterized property testers. Manuscript.

[43] M.-S. Chang, C.-C. Lin, and P. Rossmanith: New fixed-parameter algorithms for the minimum quartet inconsistency problem. *Theory Comput. Syst.* **47** (2010) 342–368.

[44] M.-S. Chang, C.-C. Lin, and P. Rossmanith: A property tester for tree-likeness of quartet topologies. *Theory Comput. Syst.* **49** (2011) 576–587.

[45] M.-S. Chang, C.-C. Lin, and P. Rossmanith: Testing tree-consistency of quartet topologies with a bounded number of missing quartets. Manuscript. A preliminary version, entitled "Testing tree-consistency with k missing quartets" and authored by the second author, appeared in the *Proceedings of the 28th Workshop on Combinatorial Mathematics and Computation Theory*, Penghu, Taiwan, 27–28 May, 2011, pp. 280–285.

[46] B. Chor: From quartets to phylogenetic trees. In *Proceedings of the 25th Conference on Current Trends in Theory and Practice of Informatics* (SOFSEM 1998), Lecture Notes in Comput. Sci., Vol. 1521, Springer, Berlin, 1998, pp. 36–53.

[47] J. Chen, I. A. Kanj, and W. Jia: Vertex cover: further observations and further improvements. *J. Algorithms* **41** (2001) 280–301.

[48] J. Chen, I. A. Kanj, and G. Xia: Improved Parameterized Upper Bounds for Vertex Cover. *Theoret. Comput. Sci.* **411** (2010) 3736–3756.

[49] J. Chen, L. Liu, and W. Jia: Improvement on vertex cover for low degree graphs. *Networks* **35** (2000) 253–259.

[50] H. Colonius and H. H. Schulze: Tree structures for proximity data. *British J. Math. Statist. Psycho.* **34** (1981) 167–180.

[51] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein: *Introduction to Algorithms, Second Edition.* MIT Press, 2001.

[52] B. Courcelle: The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inform. and Comput.* **85** (1990) 12–75.

[53] A. Czumaj, A. Shapira, and C. Sohler: Testing hereditary properties of non-expanding bounded-degree graphs. *SIAM J. Comput.* **38** (2009) 2499–2510.

[54] A. Czumaj, A. Shapira, and C. Sohler: Testing hereditary properties of non-expanding bounded-degree graphs. *SIAM J. Comput.* **38** (2009) 2499–2510.

[55] A. Czygrinow, M. Hańćkowiak, and W. Wawrzyniak: Fast distributed approximations in planar graphs. In *Proceedings of the 22nd International Symposium on Distributed Computing* (DISC 2008), Lecture Notes in Comput. Sci., Vol. 5218, Springer-Verlag, 2008, pp. 78–92.

[56] E. D. Demaine, M. T. Hajiaghayi, and K. Kawarabayashi: Algorithmic graph minor theory: Decomposition, approximation, and coloring. In *Proceedings of the 46th Annual Symposium on Foundations of Computer Science* (FOCS 2005), pp. 637–646.

[57] R. Diestel: *Graph Theory.* 4th Edition. Springer-Verlag, Graduate Texts in Mathematics, Vol. 173, 2010.

[58] I. Dinur: The **PCP** Theorem by Gap Amplification. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing* (STOC 2006), pp. 241–250.

[59] I. Dinur and S. Safra: On the hardness of approximating minimum vertex cover. *Annals of Mathematics* **162** (2005) 439–486.

[60] R. G. Downey and M. R. Fellows: *Parameterized Complexity.* Springer-Verlag, New York, 1999.

[61] V. Dujmović and D. R. Wood: Graph treewidth and geometric thickness parameters. *Discrete Comput. Geom.* **37** (2007) 641–670.

[62] P. Erdős, M. Steel, L. Székely, and T. Warnow: A few logs suffice to build (almost) all trees (Part 1). *Random Struct. Alg.* **14** (1999) 153–184.

[63] F. Ergün, S. Kannan, R. Kumar, R. Rubinfeld, and M. Vishwanathan: Spot-Checkers. *J. Comput. System Sci.* **60** (2000) 717–751.

[64] F. Ergün, R. Kumar, and R. Rubinfeld: Fast approximate probabilistic checkable proofs. *Inform. and Comput.* **189** (2004) 135–159.

[65] T. Erlebach, T. Hagerup, K. Jansen, M. Minzlaff, and A. Wolff: Trimming of graphs, with application to point labeling. In S. Albers and P. Weil, editors, *Proceedings of the 25th Symposium on Theoretical Aspects of Computer Science* (STACS 2008), volume 08001 of *Dagstuhl Seminar Proceedings*, pp. 265–276. Internationales Begegnugs - und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[66] J. Felsenstein: *Inferring phylogenies.* Sinauer Associates, Inc. Publishers. Sunderland, Massachusetts.

[67] E. Fischer: The art of uninformed decisions: A primer to property testing. The Computational Complexity Column of the *Bulletin of the European Association for Theoretical Computer Science* (BEATCS) **75** (2001) 97–126.

[68] E. Fischer, E. Lehman, I. Newman, S. Raskhodnikova, R. Rubinfeld, and A. Samorodnitsky: Monotonicity testing over general poset domains. In *Proceedings of the 34th Annual ACM Symposium on the Theory of Computing* (STOC 2002), pp. 474–483.

[69] E. Fischer and I. Newman: Testing versus estimation of graph properties. *SIAM J. Comput.* **37** (2007) 482–501.

[70] E. Fischer, I. Newman, and J. Sgall: Functions that have read-twice constant width branching programs are not necessarily testable. *Random Structures Algorithms* **24** (2004) 175–193.

[71] E. Fischer and O. Yahalom: Testing convexity properties of tree colorings. *Algorithmica* **60** (2011) 766–805.

[72] M. R. Garey and D. S. Johnson: *Computers and Intractability: A Guide to the Theory of* **NP**-*Completeness.* W. H. Freeman and Company, 1979.

[73] L. Gąsieniec, J. Jansson, A. Lingas, and A. Östlin: On the complexity of constructing evolutionary trees. *J. Comb. Optim.* **3** (1999) 183–197.

[74] O. Goldreich: "Combinatorial property testing (a survey)," Randomization methods in algorithm design (Eds., P. Pardalos, S. Rajaseekaran, and J. Rolin), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS, Providence, Vol. 43, 1998, pp. 45–59.

[75] O. Goldreich, S. Goldwasser, and D. Ron: Property testing and its connection to learning and approximation. *J. ACM* **45** (1998) 653–750.

[76] O. Goldreich and D. Ron: Property testing in bounded degree graphs. *Algorithmica* **32** (2002) 302–343.

[77] M. C. Golumbic: *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[78] J. Gramm and R. Niedermeier: A fixed-parameter algorithm for minimum quartet inconsistency. *J. Comput. System Sci.* **67** (2003) 723–741.

[79] J. Gramm, R. Niedermeier, and P. Rossmanith: Fixed-parameter algorithms for Closest String and related problems. *Algorithmica* **37** (2003) 25–42.

[80] D. H. Greene, D. E. Knuth: *Mathematics for the Analysis of Algorithms*, 3nd edn., Progress in Computer Science, Birkhäuser, Boston, MA, 1990.

[81] S. Guillemot and M. Mnich: Kernel and fast algorithm for dense triplet inconsistency. In *Proceedings of the 7th Annual Conference on Theory and Applications of Model of Computation* (TAMC 2010), Lecture Notes in Comput. Sci., Vol. 6108, Springer-Verlag, 2010, pp. 247–257.

[82] S. Halevy, O. Lachish, I. Newman, and D. Tsur: Testing properties of constraint-graphs. In *Proceedings of the 22nd IEEE Annual Conference on Computational Complexity* (CCC 2007), pp. 264–277.

[83] A. Hassidim, J. A. Kelner, H. N. Nguyen, and K. Onak: Local graph partitions for approximation and testing. *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science* (FOCS 2009), pp. 22–31.

[84] M. R. Henzinger, V. King, and T. Warnow: Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica* **24** (1999) 1–13.

[85] T. Jiang, P. E. Kearney, and M. Li: Some open problems in computational molecular biology. *J. Algorithms* **34** (2000) 194–201.

[86] T. Jiang, P. E. Kearney, and M. Li: A polynomial time approximation scheme for inferring evolutionary tree from quartet topologies and its application. *SIAM J. Comput.* **30** (2001) 1942–1961.

[87] T. Kloks: *Treewidth. Computations and Approximations.* Lecture Notes in Comput. Sci., Vol. 842, Springer-Verlag, 1994.

[88] J. Kneis, D. Mölle, S. Richter, P. Rossmanith: Divide-and-color. In *Proceedings of the 32nd International Workshop on Graph-Theoretic Concepts in Computer Science* WG 2006, Lecture Notes in Comput. Sci., Vol. 4271, Springer-Verlag, 2006, pp. 58–67.

[89] C.-C. Lin: A program computing the branching number of a branching vector. `http://www.cs.ccu.edu.tw/~lincc/Program/fbr.exe`
(The source code: `http://www.cs.ccu.edu.tw/~lincc/Program/fbr.c`)

[90] K. Mehlhorm: *Data Structures and Algorithms, Volume 2:* **NP**-*Completeness and Graph Algorithms.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1984.

[91] C. St.J. A. Nash-Williams: Decomposition of finite graphs into forests. *J. London Math. Soc.* **39** (1964) p. 12.

[92] I. Newman: Testing of functions that have small width branching programs. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science* (FOCS 2000), 2000, pp. 251–258.

[93] I. Newman: Testing of function that have small width branching programs. *SIAM J. Computing* **31** (2002) 1557–1570.

[94] I. Newman: Property testing of massively parametrized problems - A survey. *Property Testing - Current Research and Surveys*, Lecture Notes in Comput. Sci., Vol. 6390, Springer-Verlag, 2010, pp. 142–157.

[95] I. Newman and C. Sohler: Every property of hyperfinite graphs is testable. In *Proceedings of the 43th Annual ACM Symposium on Theory of Computing* (STOC 2011). Accepted.

[96] M. P. Ng and N. C.Wormald: Reconstruction of rooted trees from subtrees. *Discrete Appl. Math.* **69** (1996) 19–31.

[97] H. N. Nguyen and K. Onak: Constant-time approximation algorithms via local improvements. *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science* (FOCS 2008), pp. 327–336.

[98] R. Niedermeier: *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[99] R. Niedermeier and P. Rossmanith: Upper bounds for vertex cover further improved. In *Proceedings of the 16th International Symposium on Theoretical Aspects of Computer Science* (STACS 1999), Lecture Notes in Comput. Sci., Vol. 1563, Springer-Verlag, 1999, pp. 561–570.

[100] R. Niedermeier and P. Rossmanith: A general method to speed up fixed-parameter-tractable algorithms. *Inform. Process. Lett.* **73** (2000) 125–129.

[101] R. Niedermeier and P. Rossmanith: On efficient fixed-parameter algorithms for weighted vertex cover. *J. Algorithms* **47** (2003) 63–77.

[102] K. Onak: *New sublinear methods in the struggle against classical problems.* Ph.D. Thesis. Massachusetts Institute of Technology, 2010.

[103] M. Parnas and D. Ron: Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms. *Theoret. Comput. Sci.* **381** (2007) 183–196.

[104] M. Parnas, D. Ron and R. Rubinfeld: Tolerant property testing and distance approximation, *J. Comput. System Sci.* **72** (2006) 1012–1042.

[105] N. Robertson and P. D. Seymour: Graph minors. II: Algorithmic aspects of tree-width. *J. Algorithms* **7** (1986) 309–322.

[106] D. Ron: *Property testing*, in Handbook of Randomized Computing, Vol. II. S. Rajasekaran, P. M. Pardalos, J. H. Reif, and J. D. P. Rolim, eds., Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001, 597–649.

[107] N. Robertson and P. D. Seymour: Graph minors, XX. Wagner's conjecture. *J. Combin. Theory Ser. B* **92** (2004) 325–357.

[108] R. Rubinfeld and M. Sudan: Robust characterization of polynomials with applications to program testing. *SIAM J. Comput.* **25** (1996) 252–271.

[109] S. Snir, T. Warnow, and S. Rao: Short quartet puzzling: A new quartet-based phylogeny reconstruction algorithm. *J. Comput. Biol.* **15** (2008) 91–103.

[110] M. Steel: The complexity of reconstructing trees from qualitative characters and subtrees. *J. Classification* **9** (1992) 91–116.

[111] U. Stege and M. Fellows: An improved fixed-parameter-tractable algorithm for vertex cover. Technical Report 318, Department of Computer Science, ETH Zürich, April 1999.

[112] K. Strimmer and A. von Haeseler: Quartet puzzling: A quartet maximum-likelihood method for reconstructing tree topologies. *Mol. Biol. Evol.* **13** (1996) 964–969.

[113] E. Szemerédi: Regular partitions of graphs. In Proc. Colloque. Inter. CNRS (J. C. Bermond, J. C. Fournier, M. Las Vergnas, and Sotteau eds.), 1978, pp. 399–401.

[114] R. W. H. Verwer and J. V. Pelt: Analysis of binary trees when occasional multifurcations can be considered as aggregates of bifurcations. *Bull. Math. Biol.* **52** (1990) 629–641.

[115] B. Y. Wu: Constructing the maximum consensus tree from rooted triples. *J. Comb. Optim.* **8** (2004) 29–39.

[116] G. Wu, J-H. You, and G. Lin: A lookahead branch-and-bound algorithm for the maximum quartet consistency problem. In *Proceedings of 5th Workshop on Algorithms in Bioinformatics* (WABI 2005), Lecture Notes in Comput. Sci., Vol. 3692, Springer-Verlag, 2005, pp. 65–76.

[117] G. Wu, J-H. You, and G. Lin: A polynomial time algorithm for the minimum quartet inconsistency problem with $O(n)$ quartet errors. *Inform. Process. Lett.* **100** (2006) 167–171.

[118] Y. Yoshida and H. Ito: Property testing on k-vertex-connectivity of graphs. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming* (ICALP 2008), Lecture Notes in Comput. Sci., Vol. 5125, Springer-Verlag, 2008, pp. 539–550.

Appendix A

Fundamental Notions on Graphs

Basic definitions of graphs. A graph is a pair $G = (V, E)$ of sets such that $E \subseteq V \times V$. Thus, E consists of 2-element subsets of V. The elements of V are the *vertices* and the elements of E are the *edges* of the graph G. Sometimes we also denote by $V(G)$ and $E(G)$ the vertex set and the edge set of G, respectively. In this dissertation, graphs are always finite (i.e., V and E are finite) and simple (i.e., no two elements of E are equal). An edge of an undirected graph with endpoints u and v is denoted by (u, v) or (v, u), while in a directed graph, an directed edge from u to v is always denoted by (u, v). The endpoints of an edge $(u, v) \in E$ are said to be *adjacent*, and one is said to be a *neighbor* of the other. We say that a vertex v is *incident* with an edge $e \in E$ or e is incident to v if v is an endpoint of e. For a graph $G = (V, E)$, we set $|V| = n$ and $|E| = m$ unless they are specified otherwise. Let $\deg_G(v)$ be the number of edges incident to v in the graph G, that is, the *vertex degree* of v in G). Let $N_G(v) = \{u \in V \mid (u, v) \in E\}$ denote the set of vertices adjacent to v (i.e., the *open neighborhood* of v) in G . For a subset $V' \subseteq V$, we define $N_G(V') = \{u \in V \setminus V' \mid \exists v \in V', (u, v) \in E\}$. Let $N_G[v] = N_G(v) \cup \{v\}$ and $N_G[V'] = N_G(V') \cup V'$ denote the *closed neighborhood* of v and V' respectively. We say that a vertex is *isolated* if its vertex degree is 0. A graph $G = (V, E)$ is *empty* if $E = \emptyset$. Given a graph $G = (V, E)$, the *complement* of G is $\bar{G} = (V, V \times V \setminus E)$.

Subgraphs and induced subgraphs. For two graphs G' and G with $V' = V(G') \subseteq V = V(G)$ and $E' = E(G') \subseteq E = E(G)$, we say that G' is a *subgraph* of G. Furthermore, if G' is a subgraph of G and G' contains all the edges $(u, v) \in E(G)$ with $u, v \in V(G')$, we say that G' is an *induced subgraph* of G or the subgraph of G induced by V'. For a subset $S \subset V$, we denote by $G[S]$ the subgraph of G induced by S. With a slight abuse of notation, we denote by $G - S$ the subgraph of G induced by $V \setminus S$. A graph H is said to be a *minor* of a graph G if H can be obtained from G using a sequence of vertex removals, edge removals and edge contractions. For example, in Figure A, (b) is a minor of (a), and (c) is a minor of (b). Note that both (b) and (c) are minors of (a). It is easy to see that any subgraph of G is also a minor of G.

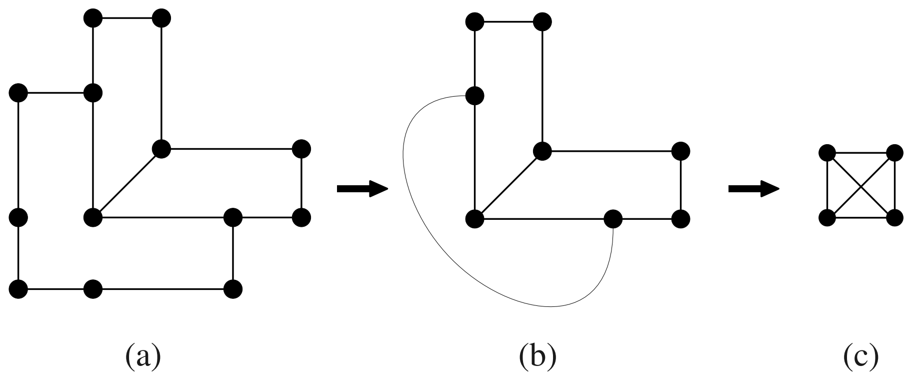

Figure A.1: Minors of a graph. (b) can be obtained by a series of vertex removals, edge removals, and edge contractions on (a). (c) can be obtained by several edge contractions on (b).

Paths, cycles, distance, and connected components. A *path* of length k is a non-empty graph $P = (V, E)$ with $V = \{v_0, v_1, \ldots, v_k\}$, $E = \{(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\}$, and all $v_i \neq v_j$ for any $0 \leq i, j \leq k$, $i \neq j$. A path is *simple* if it has no repeated vertices. A *cycle* of length k is obtained from a path P of length $k - 1$ by adding an edge (v_k, v_0). Similarly, a cycle is *simple* if it has no repeated vertices. We denote by $d(u, v)$ the *distance* between u and v in the graph G, which is the shortest length of a path between u and v. The greatest distance between any two vertices in G is the *diameter*. A graph is said to be *connected* if there is a path between every pair of vertices u and v in the graph, otherwise it is said to be *disconnected*. A *tree* is a connected graph without any cycle as its subgraph. A *connected component* C of a graph G is a connected subgraph of G with maximal size, i.e., adding any vertex $v \in V(G) \setminus V(C)$ results in a disconnected subgraph of G. G is called *k-connected* for an integer $k > 0$ if $|V(G)| > k$ and $G - X$ is still connected for every subset $X \subseteq V$ with $|X| < k$.

Independent sets and cliques. An subset $I \subseteq V$ of a graph $G = (V, E)$ is called an *independent set* if u and v are not adjacent for each pair of vertices in I. A *clique* C of a graph $G = (V, E)$ is a subset of V such that vertices in C are pairwise adjacent.

Graph coloring. A *coloring* of a graph $G = (V, E)$ is a map $f : V \mapsto S$ such that $f(u) \neq f(v)$ whenever $(u, v) \in E$. The elements in S are called *colors*. We call f a *k-coloring* of G if $|S| = k$. We say that G is *k-colorable* if it admits a k-coloring. The *chromatic number* of a graph G is the minimum integer k such that G is k-colorable.

Matchings. A *matching* M of a graph $G = (V, E)$ is a subset of E such that no two edges in M share a common endpoint.

Monotone and hereditary graph properties. A graph property can be regarded as a set of graphs. We say that a graph property $\mathcal{P}$ is *hereditary* if for each $G \in \mathcal{P}$, every induced subgraph G' of G is still in $\mathcal{P}$. A graph property $\mathcal{P}$ is *monotone* if for each $G \in \mathcal{P}$, every subgraph G' of G is still in $\mathcal{P}$. Equivalently, a graph property $\mathcal{P}$ is hereditary if removing any vertex from a graph that satisfying $\mathcal{P}$ results a graph that still satisfies $\mathcal{P}$, while a graph property $\mathcal{P}$ is monotone if removing any vertex or any edge from a graph satisfying $\mathcal{P}$ results a graph that still satisfies $\mathcal{P}$.

Hyperfinite graphs. A graph G is called (ϵ, k)-*hyperfinite* if one can remove at most ϵn edges from G to obtain a graph which has connected components of size bounded by k. For a function $\rho : \mathbb{R}^+ \mapsto \mathbb{R}+$, a collection $\mathcal{H}$ of graphs is called *ρ-hyperfinite* if every graph in $\mathcal{H}$ is $(\epsilon, \rho(\epsilon))$-hyperfinite for every $\epsilon > 0$.

We refer to the textbooks or monographs, such as [22, 34, 57, 77], for more information on graph theory.

Appendix B

Selected Probabilistic Equations and Inequalities

Here we give the probabilistic inequalities used in this dissertation. We focus on discrete probabilities and discrete random variables. We denote by $\mathbf{Pr}[E]$ the probability of an event E, where $\mathbf{Pr}[\cdot]$ denotes the probability function. The expectation of a discrete random variable X is $\mathbf{E}[X] = \sum_i i \cdot \mathbf{Pr}[X = i]$, where the summation is over all values in the range of X.

The union bound. For any finite or countably infinite sequence of events $E_1, E_2, \ldots$,

$$\mathbf{Pr}\left[\bigcup_{i\geq 1} E_i\right] \leq \sum_{i\geq 1} \mathbf{Pr}[E_i].$$

Mutually independent events. Events $E_1, E_2, \ldots, E_n$ are *mutually independent* if and only if for any subset $I \subseteq \{1, 2, \ldots, n\}$,

$$\mathbf{Pr}\left[\bigcap_{i\in I} E_i\right] = \prod_{i\in I} \mathbf{Pr}[E_i].$$

Linearity of expectations. For any finite collection of discrete random variables $X_1, X_2, \ldots, X_n$ with finite expectations,

$$\mathbf{E}\left[\sum_{i=1}^{n} a_i X_i\right] = \sum_{i=1}^{n} a_i \mathbf{E}[X_i],$$

where $a_1, a_2, \ldots, a_n$ are real numbers.

Expectations of geometric random variables. A *geometric* random variable X with parameter p is given by the following probability distribution on $n = 1, 2, \ldots$:

$$\mathbf{Pr}[X = n] = (1-p)^{n-1}p.$$

Furthermore, the expectation of X is $\mathbf{E}[X] = 1/p$.

Markov's inequality. Let X be a nonnegative random variable. Then for any $a > 0$,

$$\mathbf{Pr}[X \geq a] \leq \frac{\mathbf{E}[X]}{a}.$$

Chernoff bounds. Let $X_1, \ldots, X_n$ be mutually independent 0–1 random variables such that $\mathbf{Pr}[X_i] = p_i$. Let $S = \sum_{i=1}^{n} X_i$ and $\mu = \mathbf{E}[S]$. Then the following inequalities holds.

- for any $\delta > 0$,
$$\mathbf{Pr}[S \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu};$$
- for any $0 < \delta < 1$,
$$\mathbf{Pr}[S \geq (1+\delta)\mu] \leq e^{-\mu\delta^2/3},$$
and
$$\mathbf{Pr}[S \leq (1-\delta)\mu] \leq e^{-\mu\delta^2/2}.$$

Appendix C

Branching Vectors and Branching Numbers for FPA1-MQI

We list all the possible branching vectors as well as the corresponding branching numbers for Algorithm FPA1-MQI in Tables C.1–C.3. Note that we abbreviate topology vectors, branching vectors and branching numbers to be t.v., b.v., and b.n. respectively, and NB means there is no branching for the topology vector.

t.v.	**b.v.**	**b.n.**	**t.v.**	**b.v.**	**b.n.**
(0, 0, 0, 0, 0)	NB	NB	(0, 0, 0, 0, 1)	(1, 3, 3, 5, 5, 1, 3, 3, 3, 4, 2, 4, 4, 4, 5)	3.04454
(0, 0, 0, 0, 2)	(1, 3, 3, 5, 5, 2, 4, 4, 4, 5, 1, 3, 3, 3, 4)	3.04454	(0, 0, 0, 1, 0)	(1, 3, 3, 3, 4, 1, 3, 3, 5, 5, 2, 4, 4, 5, 4)	3.04454
(0, 0, 0, 1, 1)	NB	NB	(0, 0, 0, 1, 2)	(2, 4, 4, 4, 5, 1, 3, 3, 5, 5, 1, 3, 3, 4, 3)	3.04454
(0, 0, 0, 2, 0)	(1, 3, 3, 4, 3, 2, 4, 4, 5, 4, 1, 3, 3, 5, 5)	3.04454	(0, 0, 0, 2, 1)	(2, 4, 4, 5, 4, 1, 3, 3, 4, 3, 1, 3, 3, 5, 5)	3.04454
(0, 0, 0, 2, 2)	NB	NB	(0, 0, 1, 0, 0)	(1, 3, 3, 3, 4, 3, 3, 4, 4, 5, 3, 4, 3, 3, 4)	2.55234
(0, 0, 1, 0, 1)	(2, 4, 4, 4, 5, 2, 2, 3, 3, 4, 3, 4, 3, 3, 4)	2.46596	(0, 0, 1, 0, 2)	(2, 4, 4, 4, 5, 3, 3, 4, 4, 5, 2, 3, 2, 2, 3)	2.54314
(0, 0, 1, 1, 0)	(2, 4, 4, 2, 4, 2, 2, 3, 5, 5, 3, 4, 3, 4, 3)	2.54314	(0, 0, 1, 1, 1)	(3, 5, 5, 3, 5, 1, 1, 2, 4, 4, 3, 4, 3, 4, 3)	3.04454
(0, 0, 1, 1, 2)	(3, 5, 5, 3, 5, 2, 2, 3, 5, 5, 2, 3, 2, 3, 2)	2.67102	(0, 0, 1, 2, 0)	(2, 4, 4, 3, 3, 3, 3, 4, 5, 4, 2, 3, 2, 4, 4)	2.46596
(0, 0, 1, 2, 1)	(3, 5, 5, 4, 4, 2, 2, 3, 4, 3, 2, 3, 2, 4, 4)	2.54314	(0, 0, 1, 2, 2)	(3, 5, 5, 4, 4, 3, 3, 4, 5, 4, 1, 2, 1, 3, 3)	3.04454
(0, 0, 2, 0, 0)	(1, 3, 3, 4, 3, 3, 4, 3, 3, 4, 3, 3, 4, 4, 5)	2.55234	(0, 0, 2, 0, 1)	(2, 4, 4, 5, 4, 2, 3, 2, 2, 3, 3, 3, 4, 4, 5)	2.54314
(0, 0, 2, 0, 2)	(2, 4, 4, 5, 4, 3, 4, 3, 3, 4, 2, 2, 3, 3, 4)	2.46596	(0, 0, 2, 1, 0)	(2, 4, 4, 3, 3, 2, 3, 2, 4, 4, 3, 3, 4, 5, 4)	2.46596
(0, 0, 2, 1, 1)	(3, 5, 5, 4, 4, 1, 2, 1, 3, 3, 3, 3, 4, 5, 4)	3.04454	(0, 0, 2, 1, 2)	(3, 5, 5, 4, 4, 2, 3, 2, 4, 4, 2, 2, 3, 4, 3)	2.54314
(0, 0, 2, 2, 0)	(2, 4, 4, 4, 2, 3, 4, 3, 4, 3, 2, 2, 3, 5, 5)	2.54314	(0, 0, 2, 2, 1)	(3, 5, 5, 5, 3, 2, 3, 2, 3, 2, 2, 2, 3, 5, 5)	2.67102
(0, 0, 2, 2, 2)	(3, 5, 5, 5, 3, 3, 4, 3, 4, 3, 1, 1, 2, 4, 4)	3.04454	(0, 1, 0, 0, 0)	(1, 1, 2, 4, 4, 3, 5, 5, 3, 5, 3, 4, 3, 3, 4)	3.04454
(0, 1, 0, 0, 1)	(2, 2, 3, 5, 5, 2, 4, 4, 2, 4, 3, 4, 3, 3, 4)	2.54314	(0, 1, 0, 0, 2)	(2, 2, 3, 5, 5, 3, 5, 5, 3, 5, 2, 3, 2, 2, 3)	2.67102
(0, 1, 0, 1, 0)	(2, 2, 3, 3, 4, 2, 4, 4, 4, 5, 3, 4, 3, 4, 3)	2.46596	(0, 1, 0, 1, 1)	(3, 3, 4, 4, 5, 1, 3, 3, 3, 4, 3, 4, 3, 4, 3)	2.55234
(0, 1, 0, 1, 2)	(3, 3, 4, 4, 5, 2, 4, 4, 4, 5, 2, 3, 2, 3, 2)	2.54314	(0, 1, 0, 2, 0)	(2, 2, 3, 4, 3, 3, 5, 5, 4, 4, 2, 3, 2, 4, 4)	2.54314
(0, 1, 0, 2, 1)	(3, 3, 4, 5, 4, 2, 4, 4, 3, 3, 2, 3, 2, 4, 4)	2.46596	(0, 1, 0, 2, 2)	(3, 3, 4, 5, 4, 3, 5, 5, 4, 4, 1, 2, 1, 3, 3)	3.04454
(0, 1, 1, 0, 0)	(2, 2, 3, 3, 4, 4, 4, 5, 3, 5, 4, 4, 2, 2, 3)	2.54314	(0, 1, 1, 0, 1)	(3, 3, 4, 4, 5, 3, 3, 4, 2, 4, 4, 4, 2, 2, 3)	2.46596
(0, 1, 1, 0, 2)	(3, 3, 4, 4, 5, 4, 4, 5, 3, 5, 3, 3, 1, 1, 2)	3.04454	(0, 1, 1, 1, 0)	(3, 3, 4, 2, 4, 3, 3, 4, 4, 5, 4, 4, 2, 3, 2)	2.46596
(0, 1, 1, 1, 1)	(4, 4, 5, 3, 5, 2, 2, 3, 3, 4, 4, 4, 2, 3, 2)	2.54314	(0, 1, 1, 1, 2)	(4, 4, 5, 3, 5, 3, 3, 4, 4, 5, 3, 3, 1, 2, 1)	3.04454
(0, 1, 1, 2, 0)	(3, 3, 4, 3, 3, 4, 4, 5, 4, 4, 3, 3, 1, 3, 3)	2.55234	(0, 1, 1, 2, 1)	(4, 4, 5, 4, 4, 3, 3, 4, 3, 3, 3, 3, 1, 3, 3)	2.55234
(0, 1, 1, 2, 2)	NB	NB	(0, 1, 2, 0, 0)	(2, 2, 3, 4, 3, 4, 5, 4, 2, 4, 4, 3, 3, 3, 4)	2.46596
(0, 1, 2, 0, 1)	(3, 3, 4, 5, 4, 3, 4, 3, 1, 3, 4, 3, 3, 3, 4)	2.55234	(0, 1, 2, 0, 2)	(3, 3, 4, 5, 4, 4, 5, 4, 2, 4, 3, 2, 2, 2, 3)	2.54314
(0, 1, 2, 1, 0)	(3, 3, 4, 3, 3, 3, 4, 3, 3, 4, 4, 3, 3, 4, 3)	2.30042	(0, 1, 2, 1, 1)	(4, 4, 5, 4, 4, 2, 3, 2, 2, 3, 4, 3, 3, 4, 3)	2.46596
(0, 1, 2, 1, 2)	(4, 4, 5, 4, 4, 3, 4, 3, 3, 4, 3, 2, 2, 3, 2)	2.46596	(0, 1, 2, 2, 0)	(3, 3, 4, 4, 2, 4, 5, 4, 3, 3, 3, 2, 2, 4, 4)	2.46596
(0, 1, 2, 2, 1)	(4, 4, 5, 5, 3, 3, 4, 3, 2, 2, 3, 2, 2, 4, 4)	2.54314	(0, 1, 2, 2, 2)	(4, 4, 5, 5, 3, 4, 5, 4, 3, 3, 2, 1, 1, 3, 3)	3.04454
(0, 2, 0, 0, 0)	(1, 2, 1, 3, 3, 3, 5, 5, 4, 4, 3, 3, 4, 4, 5)	3.04454	(0, 2, 0, 0, 1)	(2, 3, 2, 4, 4, 2, 4, 4, 3, 3, 3, 3, 4, 4, 5)	2.46596
(0, 2, 0, 0, 2)	(2, 3, 2, 4, 4, 3, 5, 5, 4, 4, 2, 2, 3, 3, 4)	2.54314	(0, 2, 0, 1, 0)	(2, 3, 2, 2, 3, 2, 4, 4, 5, 4, 3, 3, 4, 5, 4)	2.54314
(0, 2, 0, 1, 1)	(3, 4, 3, 3, 4, 1, 3, 3, 4, 3, 3, 3, 4, 5, 4)	2.55234	(0, 2, 0, 1, 2)	(3, 4, 3, 3, 4, 2, 4, 4, 5, 4, 2, 2, 3, 4, 3)	2.46596

Table C.1: The possible branching vectors and branching numbers of Algorithm FPA1-MQI (part 1).

t.v.	b.v.	b.n.	t.v.	b.v.	b.n.
(0, 2, 0, 2, 0)	(2, 3, 2, 3, 2, 3, 5, 5, 5, 3, 2, 2, 3, 5, 5)	2.67102	(0, 2, 0, 2, 1)	(3, 4, 3, 4, 3, 2, 4, 4, 4, 2, 2, 2, 3, 5, 5)	2.54314
(0, 2, 0, 2, 2)	(3, 4, 3, 4, 3, 3, 5, 5, 5, 3, 1, 1, 2, 4, 4)	3.04454	(0, 2, 1, 0, 0)	(2, 3, 2, 2, 3, 4, 4, 5, 4, 4, 4, 3, 3, 3, 4)	2.46596
(0, 2, 1, 0, 1)	(3, 4, 3, 3, 4, 3, 3, 4, 3, 3, 4, 3, 3, 3, 4)	2.30042	(0, 2, 1, 0, 2)	(3, 4, 3, 3, 4, 4, 4, 5, 4, 4, 3, 2, 2, 2, 3)	2.46596
(0, 2, 1, 1, 0)	(3, 4, 3, 1, 3, 3, 3, 4, 5, 4, 4, 3, 3, 4, 3)	2.55234	(0, 2, 1, 1, 1)	(4, 5, 4, 2, 4, 2, 2, 3, 4, 3, 4, 3, 3, 4, 3)	2.46596
(0, 2, 1, 1, 2)	(4, 5, 4, 2, 4, 3, 3, 4, 5, 4, 3, 2, 2, 3, 2)	2.54314	(0, 2, 1, 2, 0)	(3, 4, 3, 2, 2, 4, 4, 5, 5, 3, 3, 2, 2, 4, 4)	2.54314
(0, 2, 1, 2, 1)	(4, 5, 4, 3, 3, 3, 3, 4, 4, 2, 3, 2, 2, 4, 4)	2.46596	(0, 2, 1, 2, 2)	(4, 5, 4, 3, 3, 4, 4, 5, 5, 3, 2, 1, 1, 3, 3)	3.04454
(0, 2, 2, 0, 0)	(2, 3, 2, 3, 2, 4, 5, 4, 3, 3, 4, 2, 4, 4, 5)	2.54314	(0, 2, 2, 0, 1)	(3, 4, 3, 4, 3, 3, 4, 3, 2, 2, 4, 2, 4, 4, 5)	2.46596
(0, 2, 2, 0, 2)	(3, 4, 3, 4, 3, 4, 5, 4, 3, 3, 3, 1, 3, 3, 4)	2.55234	(0, 2, 2, 1, 0)	(3, 4, 3, 2, 2, 3, 4, 3, 4, 3, 4, 2, 4, 5, 4)	2.46596
(0, 2, 2, 1, 1)	(4, 5, 4, 3, 3, 2, 3, 2, 3, 2, 4, 2, 4, 5, 4)	2.54314	(0, 2, 2, 1, 2)	(4, 5, 4, 3, 3, 3, 4, 3, 4, 3, 3, 1, 3, 4, 3)	2.55234
(0, 2, 2, 2, 0)	(3, 4, 3, 3, 1, 4, 5, 4, 4, 2, 3, 1, 3, 5, 5)	3.04454	(0, 2, 2, 2, 1)	(4, 5, 4, 4, 2, 3, 4, 3, 3, 1, 3, 1, 3, 5, 5)	3.04454
(0, 2, 2, 2, 2)	NB	NB	(1, 0, 0, 0, 0)	(1, 1, 2, 4, 4, 3, 4, 3, 3, 4, 3, 5, 5, 3, 5)	3.04454
(1, 0, 0, 0, 1)	(2, 2, 3, 5, 5, 2, 3, 2, 2, 3, 3, 5, 5, 3, 5)	2.67102	(1, 0, 0, 0, 2)	(2, 2, 3, 5, 5, 3, 4, 3, 3, 4, 2, 4, 4, 2, 4)	2.54314
(1, 0, 0, 1, 0)	(2, 2, 3, 3, 4, 2, 3, 2, 4, 4, 3, 5, 5, 4, 4)	2.54314	(1, 0, 0, 1, 1)	(3, 3, 4, 4, 5, 1, 2, 1, 3, 3, 3, 5, 5, 4, 4)	3.04454
(1, 0, 0, 1, 2)	(3, 3, 4, 4, 5, 2, 3, 2, 4, 4, 2, 4, 4, 3, 3)	2.46596	(1, 0, 0, 2, 0)	(2, 2, 3, 4, 3, 3, 4, 3, 4, 3, 2, 4, 4, 4, 5)	2.46596
(1, 0, 0, 2, 1)	(3, 3, 4, 5, 4, 2, 3, 2, 3, 2, 2, 4, 4, 4, 5)	2.54314	(1, 0, 0, 2, 2)	(3, 3, 4, 5, 4, 3, 4, 3, 4, 3, 1, 3, 3, 3, 4)	2.55234
(1, 0, 1, 0, 0)	(2, 2, 3, 3, 4, 4, 3, 3, 3, 4, 4, 5, 4, 2, 4)	2.46596	(1, 0, 1, 0, 1)	(3, 3, 4, 4, 5, 3, 2, 2, 2, 3, 4, 5, 4, 2, 4)	2.54314
(1, 0, 1, 0, 2)	(3, 3, 4, 4, 5, 4, 3, 3, 3, 4, 3, 4, 3, 1, 3)	2.55234	(1, 0, 1, 1, 0)	(3, 3, 4, 2, 4, 3, 2, 2, 4, 4, 4, 5, 4, 3, 3)	2.46596
(1, 0, 1, 1, 1)	(4, 4, 5, 3, 5, 2, 1, 1, 3, 3, 4, 5, 4, 3, 3)	3.04454	(1, 0, 1, 1, 2)	(4, 4, 5, 3, 5, 3, 2, 2, 4, 4, 3, 4, 3, 2, 2)	2.54314
(1, 0, 1, 2, 0)	(3, 3, 4, 3, 3, 4, 3, 3, 4, 3, 3, 4, 3, 3, 4)	2.30042	(1, 0, 1, 2, 1)	(4, 4, 5, 4, 4, 3, 2, 2, 3, 2, 3, 4, 3, 3, 4)	2.46596
(1, 0, 1, 2, 2)	(4, 4, 5, 4, 4, 4, 3, 3, 4, 3, 2, 3, 2, 2, 3)	2.46596	(1, 0, 2, 0, 0)	(2, 2, 3, 4, 3, 4, 4, 2, 2, 3, 4, 4, 5, 3, 5)	2.54314
(1, 0, 2, 0, 1)	(3, 3, 4, 5, 4, 3, 3, 1, 1, 2, 4, 4, 5, 3, 5)	3.04454	(1, 0, 2, 0, 2)	(3, 3, 4, 5, 4, 4, 4, 2, 2, 3, 3, 3, 4, 2, 4)	2.46596
(1, 0, 2, 1, 0)	(3, 3, 4, 3, 3, 3, 3, 1, 3, 3, 4, 4, 5, 4, 4)	2.55234	(1, 0, 2, 1, 1)	NB	NB
(1, 0, 2, 1, 2)	(4, 4, 5, 4, 4, 3, 3, 1, 3, 3, 3, 3, 4, 3, 3)	2.55234	(1, 0, 2, 2, 0)	(3, 3, 4, 4, 2, 4, 4, 2, 3, 2, 3, 3, 4, 4, 5)	2.46596
(1, 0, 2, 2, 1)	(4, 4, 5, 5, 3, 3, 3, 1, 2, 1, 3, 3, 4, 4, 5)	3.04454	(1, 0, 2, 2, 2)	(4, 4, 5, 5, 3, 4, 4, 2, 3, 2, 2, 2, 3, 3, 4)	2.54314
(1, 1, 0, 0, 0)	NB	NB	(1, 1, 0, 0, 1)	(3, 1, 3, 5, 5, 3, 4, 3, 1, 3, 4, 5, 4, 2, 4)	3.04454
(1, 1, 0, 0, 2)	(3, 1, 3, 5, 5, 4, 5, 4, 2, 4, 3, 4, 3, 1, 3)	3.04454	(1, 1, 0, 1, 0)	(3, 1, 3, 3, 4, 3, 4, 3, 3, 4, 4, 5, 4, 3, 3)	2.55234
(1, 1, 0, 1, 1)	(4, 2, 4, 4, 5, 2, 3, 2, 2, 3, 4, 5, 4, 3, 3)	2.54314	(1, 1, 0, 1, 2)	(4, 2, 4, 4, 5, 3, 4, 3, 3, 4, 3, 4, 3, 2, 2)	2.46596
(1, 1, 0, 2, 0)	(3, 1, 3, 4, 3, 4, 5, 4, 3, 3, 3, 4, 3, 3, 4)	2.55234	(1, 1, 0, 2, 1)	(4, 2, 4, 5, 4, 3, 4, 3, 2, 2, 3, 4, 3, 3, 4)	2.46596
(1, 1, 0, 2, 2)	(4, 2, 4, 5, 4, 4, 5, 4, 3, 3, 2, 3, 2, 2, 3)	2.54314	(1, 1, 1, 0, 0)	(3, 1, 3, 3, 4, 5, 4, 4, 2, 4, 5, 5, 3, 1, 3)	3.04454
(1, 1, 1, 0, 1)	(4, 2, 4, 4, 5, 4, 3, 3, 1, 3, 5, 5, 3, 1, 3)	3.04454	(1, 1, 1, 0, 2)	NB	NB
(1, 1, 1, 1, 0)	(4, 2, 4, 2, 4, 4, 3, 3, 3, 4, 5, 5, 3, 2, 2)	2.54314	(1, 1, 1, 1, 1)	(5, 3, 5, 3, 5, 3, 2, 2, 2, 3, 5, 5, 3, 2, 2)	2.67102
(1, 1, 1, 1, 2)	(5, 3, 5, 3, 5, 4, 3, 3, 3, 4, 4, 4, 2, 1, 1)	3.04454	(1, 1, 1, 2, 0)	(4, 2, 4, 3, 3, 5, 4, 4, 3, 3, 4, 4, 2, 2, 3)	2.46596
(1, 1, 1, 2, 1)	(5, 3, 5, 4, 4, 4, 3, 3, 2, 2, 4, 4, 2, 2, 3)	2.54314	(1, 1, 1, 2, 2)	(5, 3, 5, 4, 4, 5, 4, 4, 3, 3, 3, 3, 1, 1, 2)	3.04454
(1, 1, 2, 0, 0)	(3, 1, 3, 4, 3, 5, 5, 3, 1, 3, 5, 4, 4, 2, 4)	3.04454	(1, 1, 2, 0, 1)	NB	NB
(1, 1, 2, 0, 2)	(4, 2, 4, 5, 4, 5, 5, 3, 1, 3, 4, 3, 3, 1, 3)	3.04454	(1, 1, 2, 1, 0)	(4, 2, 4, 3, 3, 4, 4, 2, 2, 3, 5, 4, 4, 3, 3)	2.46596
(1, 1, 2, 1, 1)	(5, 3, 5, 4, 4, 3, 3, 1, 1, 2, 5, 4, 4, 3, 3)	3.04454	(1, 1, 2, 1, 2)	(5, 3, 5, 4, 4, 4, 4, 2, 2, 3, 4, 3, 3, 2, 2)	2.54314
(1, 1, 2, 2, 0)	(4, 2, 4, 4, 2, 5, 5, 3, 2, 2, 4, 3, 3, 3, 4)	2.54314	(1, 1, 2, 2, 1)	(5, 3, 5, 5, 3, 4, 4, 2, 1, 1, 4, 3, 3, 3, 4)	3.04454
(1, 1, 2, 2, 2)	(5, 3, 5, 5, 3, 5, 5, 3, 2, 2, 3, 2, 2, 2, 3)	2.67102	(1, 2, 0, 0, 0)	(2, 1, 1, 3, 3, 4, 5, 4, 3, 3, 4, 4, 5, 3, 5)	3.04454
(1, 2, 0, 0, 1)	(3, 2, 2, 4, 4, 3, 4, 3, 2, 2, 4, 4, 5, 3, 5)	2.54314	(1, 2, 0, 0, 2)	(3, 2, 2, 4, 4, 4, 5, 4, 3, 3, 3, 3, 4, 2, 4)	2.46596
(1, 2, 0, 1, 0)	(3, 2, 2, 2, 3, 3, 4, 3, 4, 3, 4, 4, 5, 4, 4)	2.46596	(1, 2, 0, 1, 1)	(4, 3, 3, 3, 4, 2, 3, 2, 3, 2, 4, 4, 5, 4, 4)	2.46596
(1, 2, 0, 1, 2)	(4, 3, 3, 3, 4, 3, 4, 3, 4, 3, 3, 3, 4, 3, 3)	2.30042	(1, 2, 0, 2, 0)	(3, 2, 2, 3, 2, 4, 5, 4, 4, 2, 3, 3, 4, 4, 5)	2.54314
(1, 2, 0, 2, 1)	(4, 3, 3, 4, 3, 3, 4, 3, 3, 1, 3, 3, 4, 4, 5)	2.55234	(1, 2, 0, 2, 2)	(4, 3, 3, 4, 3, 4, 5, 4, 4, 2, 2, 2, 3, 3, 4)	2.46596
(1, 2, 1, 0, 0)	(3, 2, 2, 2, 3, 5, 4, 4, 3, 3, 5, 4, 4, 2, 4)	2.54314	(1, 2, 1, 0, 1)	(4, 3, 3, 3, 4, 4, 3, 3, 2, 2, 5, 4, 4, 2, 4)	2.46596
(1, 2, 1, 0, 2)	(4, 3, 3, 3, 4, 5, 4, 4, 3, 3, 4, 3, 3, 1, 3)	2.55234	(1, 2, 1, 1, 0)	(4, 3, 3, 1, 3, 4, 3, 3, 4, 3, 5, 4, 4, 3, 3)	2.55234
(1, 2, 1, 1, 1)	(5, 4, 4, 2, 4, 3, 2, 2, 3, 2, 5, 4, 4, 3, 3)	2.54314	(1, 2, 1, 1, 2)	(5, 4, 4, 2, 4, 4, 3, 3, 4, 3, 4, 3, 3, 2, 2)	2.46596

Table C.2: The possible branching vectors and branching numbers of Algorithm FPA1-MQI (part 2).

t.v.	b.v.	b.n.	t.v.	b.v.	b.n.
(1, 2, 1, 2, 0)	(4, 3, 3, 2, 2, 5, 4, 4, 4, 2, 4, 3, 3, 3, 4)	2.46596	(1, 2, 1, 2, 1)	(5, 4, 4, 3, 3, 4, 3, 3, 3, 1, 4, 3, 3, 3, 4)	2.55234
(1, 2, 1, 2, 2)	(5, 4, 4, 3, 3, 5, 4, 4, 4, 2, 3, 2, 2, 2, 3)	2.54314	(1, 2, 2, 0, 0)	(3, 2, 2, 3, 2, 5, 5, 3, 2, 2, 5, 3, 5, 3, 5)	2.67102
(1, 2, 2, 0, 1)	(4, 3, 3, 4, 3, 4, 4, 2, 1, 1, 5, 3, 5, 3, 5)	3.04454	(1, 2, 2, 0, 2)	(4, 3, 3, 4, 3, 5, 5, 3, 2, 2, 4, 2, 4, 2, 4)	2.54314
(1, 2, 2, 1, 0)	(4, 3, 3, 2, 2, 4, 4, 2, 3, 2, 5, 3, 5, 4, 4)	2.54314	(1, 2, 2, 1, 1)	(5, 4, 4, 3, 3, 3, 3, 1, 2, 1, 5, 3, 5, 4, 4)	3.04454
(1, 2, 2, 1, 2)	(5, 4, 4, 3, 3, 4, 4, 2, 3, 2, 4, 2, 4, 3, 3)	2.46596	(1, 2, 2, 2, 0)	(4, 3, 3, 3, 1, 5, 5, 3, 3, 1, 4, 2, 4, 4, 5)	3.04454
(1, 2, 2, 2, 1)	NB	NB	(1, 2, 2, 2, 2)	(5, 4, 4, 4, 2, 5, 5, 3, 3, 1, 3, 1, 3, 3, 4)	3.04454
(2, 0, 0, 0, 0)	(1, 2, 1, 3, 3, 3, 3, 4, 4, 5, 3, 5, 5, 4, 4)	3.04454	(2, 0, 0, 0, 1)	(2, 3, 2, 4, 4, 2, 2, 3, 3, 4, 3, 5, 5, 4, 4)	2.54314
(2, 0, 0, 0, 2)	(2, 3, 2, 4, 4, 3, 3, 4, 4, 5, 2, 4, 4, 3, 3)	2.46596	(2, 0, 0, 1, 0)	(2, 3, 2, 2, 3, 2, 2, 3, 5, 5, 3, 5, 5, 5, 3)	2.67102
(2, 0, 0, 1, 1)	(3, 4, 3, 3, 4, 1, 1, 2, 4, 4, 3, 5, 5, 5, 3)	3.04454	(2, 0, 0, 1, 2)	(3, 4, 3, 3, 4, 2, 2, 3, 5, 5, 2, 4, 4, 4, 2)	2.54314
(2, 0, 0, 2, 0)	(2, 3, 2, 3, 2, 3, 3, 4, 5, 4, 2, 4, 4, 5, 4)	2.54314	(2, 0, 0, 2, 1)	(3, 4, 3, 4, 3, 2, 2, 3, 4, 3, 2, 4, 4, 5, 4)	2.46596
(2, 0, 0, 2, 2)	(3, 4, 3, 4, 3, 3, 3, 4, 5, 4, 1, 3, 3, 4, 3)	2.55234	(2, 0, 1, 0, 0)	(2, 3, 2, 2, 3, 4, 2, 4, 4, 5, 4, 5, 4, 3, 3)	2.54314
(2, 0, 1, 0, 1)	(3, 4, 3, 3, 4, 3, 1, 3, 3, 4, 4, 5, 4, 3, 3)	2.55234	(2, 0, 1, 0, 2)	(3, 4, 3, 3, 4, 4, 2, 4, 4, 5, 3, 4, 3, 2, 2)	2.46596
(2, 0, 1, 1, 0)	(3, 4, 3, 1, 3, 3, 1, 3, 5, 5, 4, 5, 4, 4, 2)	3.04454	(2, 0, 1, 1, 1)	NB	NB
(2, 0, 1, 1, 2)	(4, 5, 4, 2, 4, 3, 1, 3, 5, 5, 3, 4, 3, 3, 1)	3.04454	(2, 0, 1, 2, 0)	(3, 4, 3, 2, 2, 4, 2, 4, 5, 4, 3, 4, 3, 4, 3)	2.46596
(2, 0, 1, 2, 1)	(4, 5, 4, 3, 3, 3, 1, 3, 4, 3, 3, 4, 3, 4, 3)	2.55234	(2, 0, 1, 2, 2)	(4, 5, 4, 3, 3, 4, 2, 4, 5, 4, 2, 3, 2, 3, 2)	2.54314
(2, 0, 2, 0, 0)	(2, 3, 2, 3, 2, 4, 3, 3, 3, 4, 4, 4, 5, 4, 4)	2.46596	(2, 0, 2, 0, 1)	(3, 4, 3, 4, 3, 3, 2, 2, 2, 3, 4, 4, 5, 4, 4)	2.46596
(2, 0, 2, 0, 2)	(3, 4, 3, 4, 3, 4, 3, 3, 3, 4, 3, 3, 4, 3, 3)	2.30042	(2, 0, 2, 1, 0)	(3, 4, 3, 2, 2, 3, 2, 2, 4, 4, 4, 4, 5, 5, 3)	2.54314
(2, 0, 2, 1, 1)	(4, 5, 4, 3, 3, 2, 1, 1, 3, 3, 4, 4, 5, 5, 3)	3.04454	(2, 0, 2, 1, 2)	(4, 5, 4, 3, 3, 3, 2, 2, 4, 4, 3, 3, 4, 4, 2)	2.46596
(2, 0, 2, 2, 0)	(3, 4, 3, 3, 1, 4, 3, 3, 4, 3, 3, 3, 4, 5, 4)	2.55234	(2, 0, 2, 2, 1)	(4, 5, 4, 4, 2, 3, 2, 2, 3, 2, 3, 3, 4, 5, 4)	2.54314
(2, 0, 2, 2, 2)	(4, 5, 4, 4, 2, 4, 3, 3, 4, 3, 2, 2, 3, 4, 3)	2.46596	(2, 1, 0, 0, 0)	(2, 1, 1, 3, 3, 4, 4, 5, 3, 5, 4, 5, 4, 3, 3)	3.04454
(2, 1, 0, 0, 1)	(3, 2, 2, 4, 4, 3, 3, 4, 2, 4, 4, 5, 4, 3, 3)	2.46596	(2, 1, 0, 0, 2)	(3, 2, 2, 4, 4, 4, 4, 5, 3, 5, 3, 4, 3, 2, 2)	2.54314
(2, 1, 0, 1, 0)	(3, 2, 2, 2, 3, 3, 3, 4, 4, 5, 4, 5, 4, 4, 2)	2.54314	(2, 1, 0, 1, 1)	(4, 3, 3, 3, 4, 2, 2, 3, 3, 4, 4, 5, 4, 4, 2)	2.46596
(2, 1, 0, 1, 2)	(4, 3, 3, 3, 4, 3, 3, 4, 4, 5, 3, 4, 3, 3, 1)	2.55234	(2, 1, 0, 2, 0)	(3, 2, 2, 3, 2, 4, 4, 5, 4, 4, 3, 4, 3, 4, 3)	2.46596
(2, 1, 0, 2, 1)	(4, 3, 3, 4, 3, 3, 3, 4, 3, 3, 3, 4, 3, 4, 3)	2.30042	(2, 1, 0, 2, 2)	(4, 3, 3, 4, 3, 4, 4, 5, 4, 4, 2, 3, 2, 3, 2)	2.46596
(2, 1, 1, 0, 0)	(3, 2, 2, 2, 3, 5, 3, 5, 3, 5, 5, 5, 3, 2, 2)	2.67102	(2, 1, 1, 0, 1)	(4, 3, 3, 3, 4, 4, 2, 4, 2, 4, 5, 5, 3, 2, 2)	2.54314
(2, 1, 1, 0, 2)	(4, 3, 3, 3, 4, 5, 3, 5, 3, 5, 4, 4, 2, 1, 1)	3.04454	(2, 1, 1, 1, 0)	(4, 3, 3, 1, 3, 4, 2, 4, 4, 5, 5, 5, 3, 3, 1)	3.04454
(2, 1, 1, 1, 1)	(5, 4, 4, 2, 4, 3, 1, 3, 3, 4, 5, 5, 3, 3, 1)	3.04454	(2, 1, 1, 1, 2)	NB	NB
(2, 1, 1, 2, 0)	(4, 3, 3, 2, 2, 5, 3, 5, 4, 4, 4, 4, 2, 3, 2)	2.54314	(2, 1, 1, 2, 1)	(5, 4, 4, 3, 3, 4, 2, 4, 3, 3, 4, 4, 2, 3, 2)	2.46596
(2, 1, 1, 2, 2)	(5, 4, 4, 3, 3, 5, 3, 5, 4, 4, 3, 3, 1, 2, 1)	3.04454	(2, 1, 2, 0, 0)	(3, 2, 2, 3, 2, 5, 4, 4, 2, 4, 5, 4, 4, 3, 3)	2.54314
(2, 1, 2, 0, 1)	(4, 3, 3, 4, 3, 4, 3, 3, 1, 3, 5, 4, 4, 3, 3)	2.55234	(2, 1, 2, 0, 2)	(4, 3, 3, 4, 3, 5, 4, 4, 2, 4, 4, 3, 3, 2, 2)	2.46596
(2, 1, 2, 1, 0)	(4, 3, 3, 2, 2, 4, 3, 3, 3, 4, 5, 4, 4, 4, 2)	2.46596	(2, 1, 2, 1, 1)	(5, 4, 4, 3, 3, 3, 2, 2, 2, 3, 5, 4, 4, 4, 2)	2.54314
(2, 1, 2, 1, 2)	(5, 4, 4, 3, 3, 4, 3, 3, 3, 4, 4, 3, 3, 3, 1)	2.55234	(2, 1, 2, 2, 0)	(4, 3, 3, 3, 1, 5, 4, 4, 3, 3, 4, 3, 3, 4, 3)	2.55234
(2, 1, 2, 2, 1)	(5, 4, 4, 4, 2, 4, 3, 3, 2, 2, 4, 3, 3, 4, 3)	2.46596	(2, 1, 2, 2, 2)	(5, 4, 4, 4, 2, 5, 4, 4, 3, 3, 3, 2, 2, 3, 2)	2.54314
(2, 2, 0, 0, 0)	NB	NB	(2, 2, 0, 0, 1)	(3, 3, 1, 3, 3, 3, 3, 4, 3, 3, 4, 4, 5, 4, 4)	2.55234
(2, 2, 0, 0, 2)	(3, 3, 1, 3, 3, 4, 4, 5, 4, 4, 3, 3, 4, 3, 3)	2.55234	(2, 2, 0, 1, 0)	(3, 3, 1, 1, 2, 3, 3, 4, 5, 4, 4, 4, 5, 5, 3)	3.04454
(2, 2, 0, 1, 1)	(4, 4, 2, 2, 3, 2, 2, 3, 4, 3, 4, 4, 5, 5, 3)	2.54314	(2, 2, 0, 1, 2)	(4, 4, 2, 2, 3, 3, 3, 4, 5, 4, 3, 3, 4, 4, 2)	2.46596
(2, 2, 0, 2, 0)	(3, 3, 1, 2, 1, 4, 4, 5, 5, 3, 3, 3, 4, 5, 4)	3.04454	(2, 2, 0, 2, 1)	(4, 4, 2, 3, 2, 3, 3, 4, 4, 2, 3, 3, 4, 5, 4)	2.46596
(2, 2, 0, 2, 2)	(4, 4, 2, 3, 2, 4, 4, 5, 5, 3, 2, 2, 3, 4, 3)	2.54314	(2, 2, 1, 0, 0)	(3, 3, 1, 1, 2, 5, 3, 5, 4, 4, 5, 4, 4, 3, 3)	3.04454
(2, 2, 1, 0, 1)	(4, 4, 2, 2, 3, 4, 2, 4, 3, 3, 5, 4, 4, 3, 3)	2.46596	(2, 2, 1, 0, 2)	(4, 4, 2, 2, 3, 5, 3, 5, 4, 4, 4, 3, 3, 2, 2)	2.54314
(2, 2, 1, 1, 0)	NB	NB	(2, 2, 1, 1, 1)	(5, 5, 3, 1, 3, 3, 1, 3, 4, 3, 5, 4, 4, 4, 2)	3.04454
(2, 2, 1, 1, 2)	(5, 5, 3, 1, 3, 4, 2, 4, 5, 4, 4, 3, 3, 3, 1)	3.04454	(2, 2, 1, 2, 0)	(4, 4, 2, 1, 1, 5, 3, 5, 5, 3, 4, 3, 3, 4, 3)	3.04454
(2, 2, 1, 2, 1)	(5, 5, 3, 2, 2, 4, 2, 4, 4, 2, 4, 3, 3, 4, 3)	2.54314	(2, 2, 1, 2, 2)	(5, 5, 3, 2, 2, 5, 3, 5, 5, 3, 3, 2, 2, 3, 2)	2.67102
(2, 2, 2, 0, 0)	(3, 3, 1, 2, 1, 5, 4, 4, 3, 3, 5, 3, 5, 4, 4)	3.04454	(2, 2, 2, 0, 1)	(4, 4, 2, 3, 2, 4, 3, 3, 2, 2, 5, 3, 5, 4, 4)	2.54314
(2, 2, 2, 0, 2)	(4, 4, 2, 3, 2, 5, 4, 4, 3, 3, 4, 2, 4, 3, 3)	2.46596	(2, 2, 2, 1, 0)	(4, 4, 2, 1, 1, 4, 3, 3, 4, 3, 5, 3, 5, 5, 3)	3.04454
(2, 2, 2, 1, 1)	(5, 5, 3, 2, 2, 3, 2, 2, 3, 2, 5, 3, 5, 5, 3)	2.67102	(2, 2, 2, 1, 2)	(5, 5, 3, 2, 2, 4, 3, 3, 4, 3, 4, 2, 4, 4, 2)	2.54314
(2, 2, 2, 2, 0)	NB	NB	(2, 2, 2, 2, 1)	(5, 5, 3, 3, 1, 4, 3, 3, 3, 1, 4, 2, 4, 5, 4)	3.04454
(2, 2, 2, 2, 2)	(5, 5, 3, 3, 1, 5, 4, 4, 4, 2, 3, 1, 3, 4, 3)	3.04454			

Table C.3: The possible branching vectors and branching numbers of Algorithm FPA1-MQI (part 3).

Appendix D

Branching Vectors and Branching Numbers for FPA2-MQI

We list all the possible branching vectors as well as the corresponding branching numbers for Algorithm FPA2-MQI in Tables D.1–D.3. There are $3^9 = 19683$ possible $\{a, b\}$-reduced topology vectors of a sextet containing a, b and there are only 141 different branching numbers obtained by the program. In order to save pages, we only list these 141 different branching numbers as well as their corresponding branching vectors here. Note that we abbreviate topology vectors, branching vectors and branching numbers to be t.v., b.v., and b.n. respectively.

t.v.	**b.v.**	**b.n.**
(0, 0, 0, 0, 0, 0, 0, 0, 1)	(1, 5, 5, 9, 9, 2, 6, 6, 6, 8, 3, 7, 7, 7, 9)	2.01615
(0, 0, 0, 0, 0, 0, 0, 1, 0)	(1, 5, 5, 7, 8, 2, 6, 6, 8, 9, 3, 7, 7, 8, 8)	2.00904
(0, 0, 0, 0, 0, 0, 0, 1, 2)	(2, 6, 6, 8, 9, 2, 6, 6, 8, 9, 2, 6, 6, 7, 7)	1.89925
(0, 0, 0, 0, 0, 0, 1, 0, 0)	(1, 5, 5, 7, 8, 4, 6, 7, 7, 9, 4, 7, 6, 6, 8)	1.81753
(0, 0, 0, 0, 0, 0, 1, 0, 1)	(2, 6, 6, 8, 9, 3, 5, 6, 6, 8, 4, 7, 6, 6, 8)	1.72707
(0, 0, 0, 0, 0, 0, 1, 0, 2)	(2, 6, 6, 8, 9, 4, 6, 7, 7, 9, 3, 6, 5, 5, 7)	1.73388
(0, 0, 0, 0, 0, 0, 1, 1, 0)	(2, 6, 6, 6, 8, 3, 5, 6, 8, 9, 4, 7, 6, 7, 7)	1.72411
(0, 0, 0, 0, 0, 0, 1, 1, 1)	(3, 7, 7, 7, 9, 2, 4, 5, 7, 8, 4, 7, 6, 7, 7)	1.74034
(0, 0, 0, 0, 0, 0, 1, 1, 2)	(3, 7, 7, 7, 9, 3, 5, 6, 8, 9, 3, 6, 5, 6, 6)	1.70862
(0, 0, 0, 0, 0, 0, 1, 2, 0)	(2, 6, 6, 7, 7, 4, 6, 7, 8, 8, 3, 6, 5, 7, 8)	1.71943
(0, 0, 0, 0, 0, 0, 1, 2, 1)	(3, 7, 7, 8, 8, 3, 5, 6, 7, 7, 3, 6, 5, 7, 8)	1.69968
(0, 0, 0, 0, 0, 0, 1, 2, 2)	(3, 7, 7, 8, 8, 4, 6, 7, 8, 8, 2, 5, 4, 6, 7)	1.74161
(0, 0, 0, 0, 0, 1, 0, 0, 0)	(1, 3, 4, 8, 8, 4, 8, 8, 6, 9, 4, 7, 6, 6, 8)	1.90721
(0, 0, 0, 0, 0, 1, 0, 0, 1)	(2, 4, 5, 9, 9, 3, 7, 7, 5, 8, 4, 7, 6, 6, 8)	1.75615
(0, 0, 0, 0, 0, 1, 0, 0, 2)	(2, 4, 5, 9, 9, 4, 8, 8, 6, 9, 3, 6, 5, 5, 7)	1.76893
(0, 0, 0, 0, 0, 1, 0, 2, 0)	(2, 4, 5, 8, 7, 4, 8, 8, 7, 8, 3, 6, 5, 7, 8)	1.74980
(0, 0, 0, 0, 0, 1, 0, 2, 1)	(3, 5, 6, 9, 8, 3, 7, 7, 6, 7, 3, 6, 5, 7, 8)	1.70416
(0, 0, 0, 0, 0, 1, 0, 2, 2)	(3, 5, 6, 9, 8, 4, 8, 8, 7, 8, 2, 5, 4, 6, 7)	1.75447
(0, 0, 0, 0, 0, 1, 1, 0, 0)	(2, 4, 5, 7, 8, 5, 7, 8, 6, 9, 5, 7, 5, 5, 7)	1.69753
(0, 0, 0, 0, 0, 1, 1, 0, 1)	(3, 5, 6, 8, 9, 4, 6, 7, 5, 8, 5, 7, 5, 5, 7)	1.65103
(0, 0, 0, 0, 0, 1, 1, 0, 2)	(3, 5, 6, 8, 9, 5, 7, 8, 6, 9, 4, 6, 4, 4, 6)	1.67693
(0, 0, 0, 0, 0, 1, 1, 1, 0)	(3, 5, 6, 6, 8, 4, 6, 7, 7, 9, 5, 7, 5, 6, 6)	1.63986
(0, 0, 0, 0, 0, 1, 1, 1, 2)	(4, 6, 7, 7, 9, 4, 6, 7, 7, 9, 4, 6, 4, 5, 5)	1.64801
(0, 0, 0, 0, 0, 1, 1, 2, 0)	(3, 5, 6, 7, 7, 5, 7, 8, 7, 8, 4, 6, 4, 6, 7)	1.64700

Table D.1: The possible branching vectors and branching numbers of Algorithm FPA2-MQI (part 1).

t.v.	b.v.	b.n.
(0, 0, 0, 0, 0, 1, 1, 2, 1)	(4, 6, 7, 8, 8, 4, 6, 7, 6, 7, 4, 6, 4, 6, 7)	1.63135
(0, 0, 0, 0, 0, 1, 1, 2, 2)	(4, 6, 7, 8, 8, 5, 7, 8, 7, 8, 3, 5, 3, 5, 6)	1.68110
(0, 0, 0, 0, 0, 1, 2, 0, 0)	(2, 4, 5, 8, 7, 5, 8, 7, 5, 8, 5, 6, 6, 6, 8)	1.69101
(0, 0, 0, 0, 0, 1, 2, 0, 1)	(3, 5, 6, 9, 8, 4, 7, 6, 4, 7, 5, 6, 6, 6, 8)	1.65385
(0, 0, 0, 0, 0, 1, 2, 0, 2)	(3, 5, 6, 9, 8, 5, 8, 7, 5, 8, 4, 5, 5, 5, 7)	1.65797
(0, 0, 0, 0, 0, 1, 2, 1, 0)	(3, 5, 6, 7, 7, 4, 7, 6, 6, 8, 5, 6, 6, 7, 7)	1.62890
(0, 0, 0, 0, 0, 1, 2, 1, 1)	(4, 6, 7, 8, 8, 3, 6, 5, 5, 7, 5, 6, 6, 7, 7)	1.63569
(0, 0, 0, 0, 0, 1, 2, 1, 2)	(4, 6, 7, 8, 8, 4, 7, 6, 6, 8, 4, 5, 5, 6, 6)	1.62715
(0, 0, 0, 0, 0, 1, 2, 2, 0)	(3, 5, 6, 8, 6, 5, 8, 7, 6, 7, 4, 5, 5, 7, 8)	1.64254
(0, 0, 0, 0, 0, 1, 2, 2, 1)	(4, 6, 7, 9, 7, 4, 7, 6, 5, 6, 4, 5, 5, 7, 8)	1.63283
(0, 0, 0, 0, 0, 1, 2, 2, 2)	(4, 6, 7, 9, 7, 5, 8, 7, 6, 7, 3, 4, 4, 6, 7)	1.66287
(0, 0, 0, 0, 0, 2, 0, 0, 0)	(1, 4, 3, 7, 7, 4, 8, 8, 7, 8, 4, 6, 7, 7, 9)	1.90020
(0, 0, 0, 0, 0, 2, 0, 0, 1)	(2, 5, 4, 8, 8, 3, 7, 7, 6, 7, 4, 6, 7, 7, 9)	1.74332
(0, 0, 0, 0, 0, 2, 0, 0, 2)	(2, 5, 4, 8, 8, 4, 8, 8, 7, 8, 3, 5, 6, 6, 8)	1.75277
(0, 0, 0, 0, 0, 2, 1, 0, 0)	(2, 5, 4, 6, 7, 5, 7, 8, 7, 8, 5, 6, 6, 6, 8)	1.68337
(0, 0, 0, 0, 0, 2, 1, 0, 1)	(3, 6, 5, 7, 8, 4, 6, 7, 6, 7, 5, 6, 6, 6, 8)	1.63148
(0, 0, 0, 0, 0, 2, 1, 0, 2)	(3, 6, 5, 7, 8, 5, 7, 8, 7, 8, 4, 5, 5, 5, 7)	1.64683
(0, 0, 0, 0, 0, 2, 1, 2, 1)	(4, 7, 6, 7, 7, 4, 6, 7, 7, 6, 4, 5, 5, 7, 8)	1.62210
(0, 0, 0, 0, 0, 2, 1, 2, 2)	(4, 7, 6, 7, 7, 5, 7, 8, 8, 7, 3, 4, 4, 6, 7)	1.65861
(0, 0, 0, 0, 0, 2, 2, 0, 0)	(2, 5, 4, 7, 6, 5, 8, 7, 6, 7, 5, 5, 7, 7, 9)	1.68985
(0, 0, 0, 0, 0, 2, 2, 0, 1)	(3, 6, 5, 8, 7, 4, 7, 6, 5, 6, 5, 5, 7, 7, 9)	1.64413
(0, 0, 0, 0, 0, 2, 2, 0, 2)	(3, 6, 5, 8, 7, 5, 8, 7, 6, 7, 4, 4, 6, 6, 8)	1.64963
(0, 0, 0, 0, 0, 2, 2, 1, 2)	(4, 7, 6, 7, 7, 4, 7, 6, 7, 7, 4, 4, 6, 7, 7)	1.62625
(0, 0, 0, 0, 0, 2, 2, 2, 0)	(3, 6, 5, 7, 5, 5, 8, 7, 7, 6, 4, 4, 6, 8, 9)	1.65822
(0, 0, 0, 0, 0, 2, 2, 2, 1)	(4, 7, 6, 8, 6, 4, 7, 6, 6, 5, 4, 4, 6, 8, 9)	1.64222
(0, 0, 0, 0, 0, 2, 2, 2, 2)	(4, 7, 6, 8, 6, 5, 8, 7, 7, 6, 3, 3, 5, 7, 8)	1.67378
(0, 0, 0, 0, 1, 1, 0, 0, 0)	(2, 2, 4, 8, 8, 5, 8, 7, 5, 8, 5, 8, 7, 5, 8)	1.80618
(0, 0, 0, 0, 1, 1, 0, 0, 1)	(3, 3, 5, 9, 9, 4, 7, 6, 4, 7, 5, 8, 7, 5, 8)	1.70704
(0, 0, 0, 0, 1, 1, 1, 1, 0)	(4, 4, 6, 6, 8, 5, 6, 6, 6, 8, 6, 8, 6, 5, 6)	1.61250
(0, 0, 0, 0, 1, 1, 1, 1, 1)	(5, 5, 7, 7, 9, 4, 5, 5, 5, 7, 6, 8, 6, 5, 6)	1.61557
(0, 0, 0, 0, 1, 1, 1, 2, 0)	(4, 4, 6, 7, 7, 6, 7, 7, 6, 7, 5, 7, 5, 5, 7)	1.60907
(0, 0, 0, 0, 1, 1, 1, 2, 1)	(5, 5, 7, 8, 8, 5, 6, 6, 5, 6, 5, 7, 5, 5, 7)	1.60136
(0, 0, 0, 0, 1, 1, 1, 2, 2)	(5, 5, 7, 8, 8, 6, 7, 7, 6, 7, 4, 6, 4, 4, 6)	1.62463
(0, 0, 0, 0, 1, 2, 0, 0, 0)	(2, 3, 3, 7, 7, 5, 8, 7, 6, 7, 5, 7, 8, 6, 9)	1.75946
(0, 0, 0, 0, 1, 2, 0, 0, 1)	(3, 4, 4, 8, 8, 4, 7, 6, 5, 6, 5, 7, 8, 6, 9)	1.67266
(0, 0, 0, 0, 1, 2, 0, 0, 2)	(3, 4, 4, 8, 8, 5, 8, 7, 6, 7, 4, 6, 7, 5, 8)	1.66839
(0, 0, 0, 0, 1, 2, 1, 0, 0)	(3, 4, 4, 6, 7, 6, 7, 7, 6, 7, 6, 7, 7, 5, 8)	1.64007
(0, 0, 0, 0, 1, 2, 1, 0, 1)	(4, 5, 5, 7, 8, 5, 6, 6, 5, 6, 6, 7, 7, 5, 8)	1.60762
(0, 0, 0, 0, 1, 2, 1, 0, 2)	(4, 5, 5, 7, 8, 6, 7, 7, 6, 7, 5, 6, 6, 4, 7)	1.61154
(0, 0, 0, 0, 1, 2, 1, 1, 0)	(4, 5, 5, 5, 7, 5, 6, 6, 7, 7, 6, 7, 7, 6, 7)	1.59884
(0, 0, 0, 0, 1, 2, 1, 1, 1)	(5, 6, 6, 6, 8, 4, 5, 5, 6, 6, 6, 7, 7, 6, 7)	1.59739
(0, 0, 0, 0, 1, 2, 1, 1, 2)	(5, 6, 6, 6, 8, 5, 6, 6, 7, 7, 5, 6, 6, 5, 6)	1.58751
(0, 0, 0, 0, 1, 2, 2, 0, 0)	(3, 4, 4, 7, 6, 6, 8, 6, 5, 6, 6, 6, 8, 6, 9)	1.64951
(0, 0, 0, 0, 1, 2, 2, 0, 1)	(4, 5, 5, 8, 7, 5, 7, 5, 4, 5, 6, 6, 8, 6, 9)	1.62864
(0, 0, 0, 0, 1, 2, 2, 0, 2)	(4, 5, 5, 8, 7, 6, 8, 6, 5, 6, 5, 5, 7, 5, 8)	1.61404
(0, 0, 0, 0, 1, 2, 2, 1, 0)	(4, 5, 5, 6, 6, 5, 7, 5, 6, 6, 6, 6, 8, 7, 8)	1.60370
(0, 0, 0, 0, 1, 2, 2, 1, 1)	(5, 6, 6, 7, 7, 4, 6, 4, 5, 5, 6, 6, 8, 7, 8)	1.61402
(0, 0, 0, 0, 1, 2, 2, 1, 2)	(5, 6, 6, 7, 7, 5, 7, 5, 6, 6, 5, 5, 7, 6, 7)	1.58891
(0, 0, 0, 0, 2, 2, 0, 0, 0)	(2, 4, 2, 6, 6, 5, 7, 8, 7, 8, 5, 7, 8, 7, 8)	1.78846
(0, 0, 0, 0, 2, 2, 0, 0, 1)	(3, 5, 3, 7, 7, 4, 6, 7, 6, 7, 5, 7, 8, 7, 8)	1.67105
(0, 0, 0, 0, 2, 2, 1, 0, 0)	(3, 5, 3, 5, 6, 6, 6, 8, 7, 8, 6, 7, 7, 6, 7)	1.65456
(0, 0, 0, 0, 2, 2, 1, 0, 1)	(4, 6, 4, 6, 7, 5, 5, 7, 6, 7, 6, 7, 7, 6, 7)	1.60511
(0, 0, 0, 0, 2, 2, 1, 0, 2)	(4, 6, 4, 6, 7, 6, 6, 8, 7, 8, 5, 6, 6, 5, 6)	1.61003
(0, 0, 1, 0, 0, 0, 1, 0, 0)	(2, 6, 6, 6, 8, 5, 5, 7, 7, 9, 5, 7, 5, 5, 7)	1.67707
(0, 0, 1, 0, 0, 0, 1, 0, 1)	(3, 7, 7, 7, 9, 4, 4, 6, 6, 8, 5, 7, 5, 5, 7)	1.65557
(0, 0, 1, 0, 0, 0, 1, 0, 2)	(3, 7, 7, 7, 9, 5, 5, 7, 7, 9, 4, 6, 4, 4, 6)	1.67159
(0, 0, 1, 0, 0, 0, 1, 1, 1)	(4, 8, 8, 6, 9, 3, 3, 5, 7, 8, 5, 7, 5, 6, 6)	1.68546
(0, 0, 1, 0, 0, 0, 1, 1, 2)	(4, 8, 8, 6, 9, 4, 4, 6, 8, 9, 4, 6, 4, 5, 5)	1.67159
(0, 0, 1, 0, 0, 0, 1, 2, 1)	(4, 8, 8, 7, 8, 4, 4, 6, 7, 7, 4, 6, 4, 6, 7)	1.64932
(0, 0, 1, 0, 0, 0, 1, 2, 2)	(4, 8, 8, 7, 8, 5, 5, 7, 8, 8, 3, 5, 3, 5, 6)	1.68845
(0, 0, 1, 0, 0, 0, 2, 0, 0)	(2, 6, 6, 7, 7, 5, 6, 6, 6, 8, 5, 6, 6, 6, 8)	1.65859
(0, 0, 1, 0, 0, 0, 2, 1, 0)	(3, 7, 7, 6, 7, 4, 5, 5, 7, 8, 5, 6, 6, 7, 7)	1.63311
(0, 0, 1, 0, 0, 0, 2, 1, 1)	(4, 8, 8, 7, 8, 3, 4, 4, 6, 7, 5, 6, 6, 7, 7)	1.66127
(0, 0, 1, 0, 0, 0, 2, 1, 2)	(4, 8, 8, 7, 8, 4, 5, 5, 7, 8, 4, 5, 5, 6, 6)	1.63797
(0, 0, 1, 0, 0, 1, 1, 0, 2)	(4, 6, 7, 7, 9, 6, 6, 8, 6, 9, 5, 6, 3, 3, 5)	1.68248
(0, 0, 1, 0, 0, 1, 1, 1, 0)	(4, 6, 7, 5, 8, 5, 5, 7, 7, 9, 6, 7, 4, 5, 5)	1.62613

Table D.2: The possible branching vectors and branching numbers of Algorithm FPA2-MQI (part 2).

t.v.	b.v.	b.n.
$(0,0,1,0,0,1,1,1,1)$	$(5,7,8,6,9,4,4,6,6,8,6,7,4,5,5)$	1.63537
$(0,0,1,0,0,1,1,1,2)$	$(5,7,8,6,9,5,5,7,7,9,5,6,3,4,4)$	1.66684
$(0,0,1,0,0,1,1,2,1)$	$(5,7,8,7,8,5,5,7,6,7,5,6,3,5,6)$	1.62884
$(0,0,1,0,0,1,1,2,2)$	$(5,7,8,7,8,6,6,8,7,8,4,5,2,4,5)$	1.70414
$(0,0,1,0,0,1,2,0,0)$	$(3,5,6,7,7,6,7,7,5,8,6,6,5,5,7)$	1.62214
$(0,0,1,0,0,1,2,0,2)$	$(4,6,7,8,8,6,7,7,5,8,5,5,4,4,6)$	1.63127
$(0,0,1,0,0,1,2,1,0)$	$(4,6,7,6,7,5,6,6,6,8,6,6,5,6,6)$	1.59355
$(0,0,1,0,0,2,1,0,0)$	$(3,6,5,5,7,6,6,8,7,8,6,6,5,5,7)$	1.62469
$(0,0,1,0,0,2,1,0,1)$	$(4,7,6,6,8,5,5,7,6,7,6,6,5,5,7)$	1.60127
$(0,0,1,0,0,2,1,1,2)$	$(5,8,7,5,8,5,5,7,8,8,5,5,4,5,5)$	1.62455
$(0,0,1,0,0,2,1,2,1)$	$(5,8,7,6,7,5,5,7,7,6,5,5,4,6,7)$	1.60518
$(0,0,1,0,0,2,2,0,0)$	$(3,6,5,6,6,6,7,7,6,7,6,5,6,6,8)$	1.61396
$(0,0,1,0,0,2,2,1,0)$	$(4,7,6,5,6,5,6,6,7,7,6,5,6,7,7)$	1.59498
$(0,0,1,0,1,0,1,0,0)$	$(3,5,6,6,8,6,5,6,6,8,6,8,6,4,7)$	1.63405
$(0,0,1,0,1,0,1,1,0)$	$(4,6,7,5,8,5,4,5,7,8,6,8,6,5,6)$	1.62053
$(0,0,1,0,1,0,1,1,2)$	$(5,7,8,6,9,5,4,5,7,8,5,7,5,4,5)$	1.63274
$(0,0,1,0,1,0,1,2,1)$	$(5,7,8,7,8,5,4,5,6,6,5,7,5,5,7)$	1.61157
$(0,0,1,0,1,2,1,2,0)$	$(5,6,6,5,6,7,6,7,7,6,6,6,5,5,7)$	1.58515
$(0,0,1,0,1,2,2,1,0)$	$(5,6,6,5,6,6,6,5,6,6,7,6,7,6,7)$	1.58142
$(0,0,1,1,0,0,1,1,0)$	$(4,8,8,4,8,3,3,5,9,9,5,7,5,7,5)$	1.71452
$(0,0,1,1,0,0,1,1,1)$	$(5,9,9,5,9,2,2,4,8,8,5,7,5,7,5)$	1.82326
$(0,0,1,1,0,0,1,1,2)$	$(5,9,9,5,9,3,3,5,9,9,4,6,4,6,4)$	1.74055
$(0,0,1,1,0,0,1,2,0)$	$(4,8,8,5,7,4,4,6,9,8,4,6,4,7,6)$	1.66043
$(0,0,1,1,0,0,1,2,1)$	$(5,9,9,6,8,3,3,5,8,7,4,6,4,7,6)$	1.70234
$(0,0,1,1,0,0,1,2,2)$	$(5,9,9,6,8,4,4,6,9,8,3,5,3,6,5)$	1.71416
$(0,0,1,1,0,0,2,1,0)$	$(4,8,8,5,7,3,4,4,8,8,5,6,6,8,6)$	1.67106
$(0,0,1,1,0,0,2,1,1)$	$(5,9,9,6,8,2,3,3,7,7,5,6,6,8,6)$	1.76722
$(0,0,1,1,0,0,2,1,2)$	$(5,9,9,6,8,3,4,4,8,8,4,5,5,7,5)$	1.68861
$(0,0,1,1,0,0,2,2,1)$	$(5,9,9,7,7,3,4,4,7,6,4,5,5,8,7)$	1.67874
$(0,0,1,1,0,1,1,2,0)$	$(5,7,8,5,7,5,5,7,8,8,5,6,3,6,5)$	1.63559
$(0,0,1,1,0,1,1,2,2)$	$(6,8,9,6,8,5,5,7,8,8,4,5,2,5,4)$	1.71650
$(0,0,1,1,0,1,2,1,2)$	$(6,8,9,6,8,4,5,5,7,8,5,5,4,6,4)$	1.64208
$(0,0,1,1,0,1,2,2,1)$	$(6,8,9,7,7,4,5,5,6,6,5,5,4,7,6)$	1.62207
$(0,0,1,1,0,1,2,2,2)$	$(6,8,9,7,7,5,6,6,7,7,4,4,3,6,5)$	1.65122
$(0,0,1,1,1,0,2,1,0)$	$(5,7,8,5,7,4,4,3,7,7,6,7,7,7,6)$	1.64437
$(0,0,1,1,1,0,2,1,1)$	$(6,8,9,6,8,3,3,2,6,6,6,7,7,7,6)$	1.74882
$(0,0,1,1,1,1,2,2,0)$	$(6,6,8,6,6,6,6,5,6,6,6,6,5,6,6)$	1.58005
$(0,0,1,1,1,1,2,2,1)$	$(7,7,9,7,7,5,5,4,5,5,6,6,5,6,6)$	1.60915
$(0,0,1,2,0,0,1,2,0)$	$(4,8,8,6,6,5,5,7,9,7,3,5,3,7,7)$	1.68272
$(0,0,1,2,0,0,1,2,1)$	$(5,9,9,7,7,4,4,6,8,6,3,5,3,7,7)$	1.69959
$(0,0,1,2,0,0,1,2,2)$	$(5,9,9,7,7,5,5,7,9,7,2,4,2,6,6)$	1.80572
$(0,0,1,2,0,1,1,2,0)$	$(5,7,8,6,6,6,6,8,8,7,4,5,2,6,6)$	1.67859
$(0,0,1,2,0,1,1,2,1)$	$(6,8,9,7,7,5,5,7,7,6,4,5,2,6,6)$	1.68505
$(0,0,1,2,0,1,1,2,2)$	$(6,8,9,7,7,6,6,8,8,7,3,4,1,5,5)$	1.86175
$(0,0,1,2,0,2,1,2,2)$	$(6,9,8,6,6,6,6,8,9,6,3,3,2,6,6)$	1.75660
$(0,0,1,2,1,0,1,2,2)$	$(6,8,9,7,7,6,5,6,8,6,3,5,3,5,6)$	1.66603
$(0,0,1,2,1,1,1,2,1)$	$(7,7,9,7,7,6,5,6,6,5,5,6,3,5,6)$	1.62625
$(0,0,1,2,1,1,1,2,2)$	$(7,7,9,7,7,7,6,7,7,6,4,5,2,4,5)$	1.69519
$(0,1,1,2,0,1,1,2,0)$	$(6,6,8,6,6,7,7,9,7,7,5,5,1,5,5)$	1.77858
$(0,1,2,1,0,1,2,1,0)$	$(6,6,8,6,6,5,7,5,5,7,7,5,5,7,5)$	1.59510
$(0,1,2,1,0,1,2,1,1)$	$(7,7,9,7,7,4,6,4,4,6,7,5,5,7,5)$	1.63029

Table D.3: The possible branching vectors and branching numbers of Algorithm FPA2-MQI (part 3).

Index of Special Symbols

Index