Aspects de l'efficacité dans des problèmes sélectionnés pour des calculs sur les graphes de grande taille

Mengchuan Zou

Université de Paris

ECOLE DOCTORALE DE SCIENCES MATHEMATIQUES DE PARIS CENTRE (ED 386)

Institut de Recherche en Informatique Fondamentale (IRIF)

Aspects of Efficiency in Selected Problems of Computation on Large Graphs

Par Mengchuan ZOU

Thèse de doctorat de: Informatique

Dirigée par Adrian KOSOWSKI
Et par Michel HABIB

Présentée et soutenue publiquement le 17 December 2019

Devant un jury composé de :

Cristina BAZGAN	PR	Université Paris Dauphine	Examinatrice
Pierluigi CRESCENZI	PR	Université de Paris	Président du jury
Michel HABIB	PR	Université de Paris	Directeur
Emmanuel GODARD	PR	Université Aix-Marseille	Rapporteur
Adrian KOSOWSKI	CR	Inria & Université de Paris	Directeur
Christophe PAUL	DR	CNRS	Rapporteur

Titre: Aspects de l'efficacité dans des problèmes sélectionnés pour des calculs sur les graphes de grande taille

Résumé:

Cette thèse présente trois travaux liés à la conception d'algorithmes efficaces applicables à des graphes de grande taille.

Dans le premier travail, nous nous plaçons dans le cadre du calcul centralisé, et ainsi la question de la généralisation des décompositions modulaires et de la conception d'un algorithme efficace pour ce problème. La décomposition modulaire et la détection de module, sont des moyens de révéler et d'analyser les propriétés modulaires de données structurées. Comme la décomposition modulaire classique est bien étudiée et possède un algorithme de temps linéaire optimal, nous étudions d'abord les généralisations de ces concepts en hypergraphes. C'est un sujet peu étudié mais qui permet de trouver de nouvelles structurations dans les familles de parties. Nous présentons ici des résultats positifs obtenus pour trois définitions de la décomposition modulaire dans les hypergraphes de la littérature. Nous considérons également la généralisation en permettant des erreurs dans les modules de graphes classiques et présentons des résultats négatifs pour deux telles définitions.

Le deuxième travail est sur des requêtes de données dans un graphe. Ici, le modèle diffère des scénarios classiques dans le sens que nous ne concevons pas d'algorithmes pour résoudre un problème original, mais nous supposons qu'il existe un oracle fournissant des informations partielles sur la solution de problème initial, où les oracle ont une consommation de temps ou de ressources de requête que nous modélisons en tant que coûts, et nous avons besoin d'un algorithme décidant comment interroger efficacement l'oracle pour obtenir la solution exacte au problème initial. L'efficacité ici concerne le coût de la requête. Nous étudions un problème de la méthode de dichotomie généralisée pour lequel nous calculons une stratégie d'interrogation efficace afin de trouver une cible cachée dans le graphe. Nous présentons les résultats de nos travaux sur l'approximation de la stratégie optimale de recherche en dichotomie généralisée sur les arbres pondérés.

Notre troisième travail est sur la question de l'efficacité de la mémoire. La configuration dans laquelle nous étudions sont des calculs distribués et avec la limitation en mémoire. Plus précisément, chaque nœud stocke ses données locales en échangeant des données par transmission de messages et est en mesure de procéder à des calculs locaux. Ceci est similaire au modèle LOCAL / CONGEST en calcul distribué, mais notre modèle requiert en outre que chaque nœud ne puisse stocker qu'un nombre constant de variables w.r.t. son degré. Ce modèle peut également décrire des algorithmes naturels. Nous implémentons une procédure existante de repondération multiplicative pour approximer le problème de flux maximal sur ce modèle.

D'un point de vue méthodologique, les trois types d'efficacité que nous avons étudiées correspondent aux trois types de scénarios suivants:

– Le premier est le plus classique. Considérant un problème, nous essayons de concevoir à la main l'algorithme le plus efficace.

– Dans le second, l'efficacité est considérée comme un objectif. Nous modélisons les coûts de requête comme une fonction objectif, et utilisons des techniques d'algorithme d'approximation pour obtenir la conception d'une stratégie efficace.

– Dans le troisième, l'efficacité est en fait posée comme une contrainte de mémoire et nous concevons un algorithme sous cette contrainte.

Mots clefs : graphes de grande taille, décomposition modulaire, hypergraphes, problème de recherche, requête de données, algorithme distribué, problème de flux maximal

Title: Aspects of Efficiency in Selected Problems of Computation on Large Graphs

Abstract: This thesis presents three works on different aspects of efficiency of algorithm design for large scale graph computations.

In the first work, we consider a setting of classical centralized computing, and we consider the question of generalizing modular decompositions and designing time-efficient algorithm for this problem. Modular decomposition, and more broadly module detection, are ways to reveal and analyze modular properties in structured data. As the classical modular decomposition is well studied and have an optimal linear-time algorithm, we firstly study the generalizations of these concepts to hypergraphs and present here positive results obtained for three definitions of modular decomposition in hypergraphs from the literature. We also consider the generalization of allowing errors in classical graph modules and present negative results for two this kind of definitions.

The second work focuses on graph data query scenarios. Here the model differs from classical computing scenarios in that we are not designing algorithms to solve an original problem, but we assume that there is an oracle which provides partial information about the solution to the original problem, where oracle queries have time or resource consumption, which we model as costs, and we need to have an algorithm deciding how to efficiently query the oracle to get the exact solution to the original problem, thus here the efficiency is addressing to the query costs. We study the generalized binary search problem for which we compute an efficient query strategy to find a hidden target in graphs. We present the results of our work on approximating the optimal strategy of generalized binary search on weighted trees.

Our third work draws attention to the question of memory efficiency. The setup in which we perform our computations is distributed and memory-restricted. Specifically, every node stores its local data, exchanging data by message passing, and is able to proceed local computations. This is similar to the LOCAL/CONGEST model in distributed computing, but our model additionally requires that every node can only store a constant number of variables w.r.t. its degree. This model can also describe natural algorithms. We implement an existing procedure of multiplicative reweighting for approximating the maximum s–t flow problem on this model, this type of methodology may potentially provide new opportunities for the field of local or natural algorithms.

From a methodological point of view, the three types of efficiency concerns correspond to the following types of scenarios: the first one is the most classical one – given the problem, we try to design by hand the more efficient algorithm; the second one, the efficiency is regarded as an objective function – where we model query costs as an objective function, and using approximation algorithm techniques to get a good design of efficient strategy; the third one, the efficiency is in fact posed as a constraint of memory and we design algorithm under this constraint.

Keywords : large graphs, modular decomposition, hypergraphs, search problem, data query, distributed algorithm, max-flow problem

Acknowledgement

Firstly I will thank my PhD supervisors Adrian Kosowski and Michel Habib. Adrian is one of the most joyful person I know in the world, besides Adrian's genius and kindly supervising of PhD study, Adrian also shows me how to decide and choose to be a person upon one's free will. Thanks Adrian and Zuzanna, and two sons, Konstanty and Feliks, for bringing the joyness in my PhD study, in lack of it, it's too hard for years of PhD.

Michel, who encounters me since the MPRI course for which I got the lowest score among all my MPRI courses in M2, and surprisingly being my PhD's co-supervisor, has given many detailed guidance of PhD study, and has enormous patience for helping with different things. Discussing with Michel not only brings advancement to studies, but also improves my French! Thanks Michel for the great patience and generously help during my PhD.

Also I'd like to sincerely thank reviewers and all jury members of this thesis, thank you a lot for your time and interest of reviewing, participating the defense, and presenting your opinions of my thesis!

I will also thank all other professors I've worked with in my PhD and master's studies. Thanks to Laurent Viennot for all the support and advising in the group of Inria GANG. Thanks to Xavier Gandibleux and Florian Richoux for my study in University of Nantes. And thanks many professors in MPRI, Pierre Fraigniaud, Iordanis Kerenidis, Christiphe Dürr, ... and so on who give my favorite courses or provided PhD opportunities when I did my master. And many thanks to Sophie Laplante for supports as our study director of MPRI.

And I will thank all other collaborators worked with me during my PhD, Dariusz Dereniowski, Fabien de Montgolfier, Lalla Mouatadid, and Przemysław Uznański, for the enjoyness of working together.

Thanks for all the colleagues and classmates of my PhD at IRIF, and also the administration members of IRIF, ED386 and Inria. Again, an extra thank for your attention if you are reading this thesis.

Thanks for two of my friends, Xiaojuan Qi, for the reading and suggestions of the introduction of my thesis; and Jiayi Wu, for many on-call jokes which are important for the PhD study.

In the last, great thanks to my family for the love and support since I was formed.

Contents

Chapter 1

Introduction

1.1 Overview

In nowadays real applications, as the scale of data growing rapidly, we are usually required to deal with huge amount of data and large structures. The efficiency of algorithm draws very often a crucial point in problem-solving in these scenarios, and moreover, new aspects of efficiency, extending the classical measure with which we call "efficient algorithms" when they are in polynomial time, raise and come more and more frequently into our attentions. In this thesis, we study the algorithm design for different problems in three different aspects of efficiency requirements.

This thesis presents three works on different aspects of efficiency of algorithm design for large scale graph computations.

In the first work, we consider a setting of classical centralized computing, and we consider the question of generalizing modular decompositions and designing time-efficient algorithm for this problem. Modular decomposition, and more broadly module detection, are ways to reveal and analyze modular properties in structured data. As the classical modular decomposition is well studied and have an optimal linear-time algorithm [45], we firstly study the generalizations of these concepts to hypergraphs, of which the new ones are more complicated to analyze and provide us with new structures in organized datas. We present here positive results obtained for three definitions of modular decomposition in hypergraphs from the literature[73, 13, 26], as well as our works on their polynomial time decomposition algorithms. We also consider the generalization of allowing errors in classical graph modules and present negative results for two definitions of allowing errors in graph modules that does not satisfy the unique decomposition theorem. We present this work in Chapter 2.

The second work focuses on large scale graph data query scenarios. Here the model differs from classical computing scenarios in that we are not designing algorithms to solve an original problem, but we assume that there is an oracle which provides partial information about the solution to the original problem, where oracle queries have time or resource consumption, which we model as costs, and we need to have an algorithm deciding how to efficiently query the oracle to get the exact solution to the original problem, thus here the efficiency is addressing to the query costs. We study one problem in data querying, i.e. the generalized binary search

problem [37] for which we compute an efficient query strategy to find a hidden target in graphs. We present the results of our work on approximating the optimal strategy of generalized binary search on weighted trees. Details of this work are contained in Chapter 3.

Our third work draws attention to the question of memory efficiency. The setup in which we perform our computations is distributed and memory-restricted. Specifically, every node stores its local data, exchanging data by message passing, and is able to proceed local computations. This is similar to the LOCAL/CONGEST model in distributed computing, but our model additionally requires that every node can only store a constant number of variables w.r.t. its degree, which prohibits a general framework of algorithm design in LOCAL/CONGEST model (such as gathering the local information of all nodes into one node, then performing computations on this specific node). This model can also describe natural algorithms (computations performed by biological agents, such as the recently studied *Physarum* dynamics[5]). We design the distributed algorithm for approximating the maximum $s-t$ flow problem on this model by implementing the ideas of an algorithm based on solving Laplacian systems [20], where they provide an algorithm design framework from the view of iterative optimization and variants of gradient descent method, the specific results of the thesis make use of the procedure of multiplicative reweighting in [20], this type of methodology forms a bridge between classical algorithms and contemporary Machine Learning approaches, and may potentially provide new opportunities for the field of local or natural algorithms. This work is explained in Chapter 4.

From a methodological point of view, the three types of efficiency concerns correspond to the following types of scenarios: the first one is the most classical one – given the problem, we try to design by hand the more efficient algorithm; the second one, the efficiency is regarded as an objective function – where we model query costs as an objective function, and using approximation algorithm techniques to get a good design of efficient strategy; the third one, the efficiency is in fact posed as a constraint of memory and we design algorithm under this constraint.

In what follows, Section 1.2, 1.3 and 1.4 , each section introduces a work constituting the thesis. Section 1.5 lists the publications associated to works presenting in this thesis.

1.2 Modular Decomposition and Generalizations

1.2.1 Modular Decomposition in Graphs

Modules and modular decomposition are introduced in [41] by Gallai in 1967 initially to analyze the structure of comparability graphs, then has been used and defined in many areas of discrete mathematics, including for graphs, 2-structures, set systems, hypergraphs, clutters, boolean functions, etc.

Briefly speaking, modules describe a character that elements of a module behave exactly the same with respect to the outside of the module in a given structure. In a graph, module is defined as a set of vertices that have the same set of neighborhood outside of the module. Precisely, for an undirected graph $G = (V, \mathcal{E})$, a **module** $M \subseteq V(G)$ satisfies: $\forall x, y \in M,\ N(x) \setminus M = N(y) \setminus M$, where $N(v)$ denotes the

neighborhood of v. In other words, $M \subseteq V(G)$ is a module if and only if for all $u \in V(G) \setminus M$, either u adjacent to all elements of M or no element of M. Given a subset of vertices $C \in V(G)$, if there exists $u \in V(G) \setminus C$, and $x, y \in C$, such that $ux \in \mathcal{E}(G)$ but $uy \notin \mathcal{E}(G)$ then u is called a splitter for C.

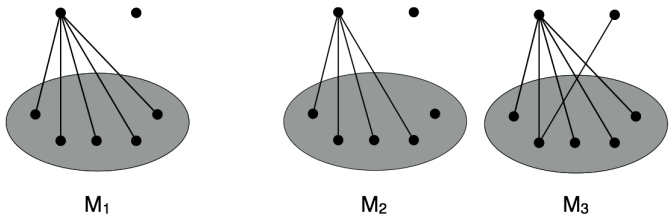


Figure 1.1: M_1 (vertices in the circle) is a module, while M_2 and M_3 are not.

Then we introduce the modular decomposition. A **strong module** is a module that does not overlap with other modules, here we say two non-empty sets A and B **overlap** if $A \cap B \neq \emptyset, A \setminus B \neq \emptyset$, and $B \setminus A \neq \emptyset$.

Definition 1.2.1. *A **modular decomposition tree** is defined as follows [13]:*

(a) tree nodes are strong modules;

(b) parent relation is the containment relation of sets represented by tree nodes;

(c) each internal nodes is labeled as

- ***complete** if the union of any subset of its children is a module;*
- ***prime** if each of its children is a module while no other union of a proper subset of its children is a module.*

If a node has only two children, to define this node to be prime or complete is equivalent, we take the convention here that a node has only two children is complete. We give an example of a graph with its modular decomposition tree [1] :

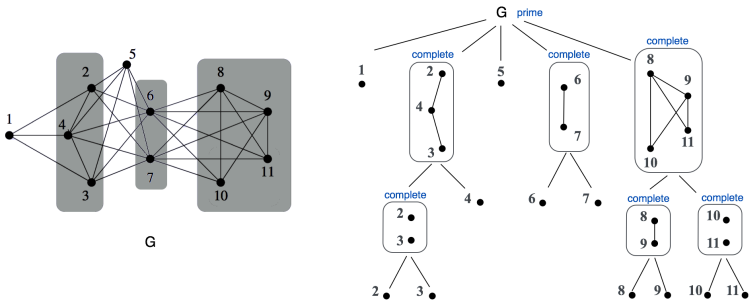


Figure 1.2: Left : A graph with its strong modules grouped. Right: The corresponding modular decomposition tree.

[1] The left example is taken from the teaching materials of Michel HABIB.

From a series of theorems of partitive family and decomposition [41, 19] that we will present in Chapter 2, it is well-known that the family of modules in a graph corresponds to an unique decomposition tree, and **modular decomposition** is the procedure to build this decomposition tree.

Modular decomposition in graphs derives several parameters and graph classes to measure the structure of a graph. For example, a graph is totally decomposable (or cograph, P_4-free graph) if there is no prime node in the decomposition tree. Cographs form a well studied graph class where many classical NP-hard problems such as maximum clique, maximum independent set, Hamiltonicity become tractable [25]. Modular decomposition has been used in fixed parameter tractable (FPT) algorithms studies, like Cluster editing [47] or modular width, a graph parameter defined with a similar decomposition procedure following the idea of modular decomposition [40].

The algorithm for modular decomposition is the basic building-block for above applications, and for graphs, it is known to have linear-time algorithms to compute a modular decomposition tree [47, 87].

Besides the study of modules in discrete structures, study of modules have appeared recently in networks in social sciences [83], and biology [35, 34], where a module is considered as a regularity or a community that has to be detected and understood.

1.2.2 Generalization of Modules

We consider here generalizing the idea of modules and modular decomposition in order to help characterize and analyze more structures in organized data. For this purpose, we are trying to find out these questions:

- How could we generalize the definition? What structures they characterize?
- Are these generalizations well-defined? (i.e. lead to an unique decomposition?)
- Could we compute generalized modular decomposition efficiently? What is the time complexity?

We then present the basic elements that decide these questions.

Partitive. Partitive is the essential property required by a valid definition for generalization of modules, because it leads to the unique decomposition. A family of subsets $\mathcal{F}$ over a ground set V is **partitive** if it satisfies the following properties [19]:

(i) $\emptyset$, V and all singletons $\{x\}$ for $x \in V$ belong to $\mathcal{F}$.

(ii) $\forall A, B \in \mathcal{F}$ that overlap, $A \cap B, A \cup B, A \setminus B$ and $A \Delta B \in \mathcal{F}$. ($\Delta$ denote the symmetric difference operation)

From [19], every partitive family has a unique decomposition tree. In our work, it's essential to check if the generalizations are partitive.

Decomposition algorithm. If the definition is valid, then we turn to the question that if there exists an efficient algorithm for decomposition. Here the algorithm takes the input (in our work it is a graph or a hypergraph), and outputs a tree-structure such that every node is a strong module w.r.t. the definition, and is labelled with either "prime" or "complete" (indicating the prime/complete node in Definition 1.2.1).

Compared to other works in this thesis, here the decomposition algorithm runs in the classical and centralized scenario and the efficiency is measured by time complexity, where we'd like to have polynomial-time algorithm. Note that even if the family has a unique decomposition tree, the unique decomposition theorem does not guarantee that the decomposition tree could be found in polynomial time, so we need to look into the definition for each generalization.

1.2.3 Related works

For graphs, the only known valid generalization is for module in directed graphs [70], they obtained a linear-time algorithm for modular decomposition in directed graphs.

For hypergraphs, we found three variations of modules: the standard modules [73], the k-subset modules [13] and the Courcelle's modules [26], each one of them leads to a unique decomposition. Only Courcelle's module is known to have linear-time decomposition algorithm [18]. For the standard module, the only known previous work is the existence of a polynomial time decomposition algorithm for clutters (a class of hypergraphs) based on its $O(n^4m^3)$ modular closure algorithm [74]. For k-subset module, the best known algorithm is not polynomial w.r.t. n and m, and is in $O(n^{3k-5})$ time [13] where k denotes the maximal size of an edge.

1.2.4 Outline of Our Work

For graphs, we have looked at ϵ-module and ϵ-splitter module for graphs, and obtained negative results of them. For hypergraphs, we developed $O(n^3 \cdot l)$ algorithms for the standard module and the k-subset module, improving the previous known $O(n^4m^3)$ algorithm in [74] and $O(n^{3k-5})$ algorithm in [13] respectively. Note that the results for k-subset module also conclude the decomposition of k-subset module in hypargraphs is in P.

Generalization in graphs. In graphs, we consider two generalizations: ϵ-module and ϵ-splitter module, ϵ-module generalizes graph modules in tolerating ϵ edges of errors per node outside the ϵ-module (not ϵ errors per module), while ϵ-splitter module tolerates errors on nodes.

Definition 1.2.2. *A subset $M \subseteq V(G)$ is an ϵ-module if $\forall x \in V(G) \setminus M$, either $|M \cap N(x)| \leq \epsilon$ or $|M \cap N(x)| \geq |M| - \epsilon$*

Definition 1.2.3. *A subset $M \subseteq V(G)$ is an ϵ-splitter module if there are at most ϵ splitters in $V(G) \setminus M$.*

We conclude in Chapter 2 that these two definitions are not partitive and one-step of parallel decomposition for ϵ-module with $\epsilon = 1$ is NP-hard.

Generalization in hypergraphs. We found in the literature three variations of modules defined in the hypergarphs or similar structures: the (we-called) standard modules defined in [73], the k-subset modules defined in [13] and the Courcelle's modules defined in [26]. And we list these three different definitions here:

Definition 1.2.4. *(**standard hypergraph module** [74, 73]) Given a hypergraph H, a **standard module** $M \subseteq V(H)$ satisfies: $\forall A, B \in \mathcal{E}(H)$ s.t. $A \cap M \neq \emptyset$, $B \cap M \neq \emptyset$ then $(A \setminus M) \cup (B \cap M) \in \mathcal{E}(H)$.*

Definition 1.2.5. *(**k-subset module** [13]) Given a hypergraph H, we call **k-subset module** $M \subseteq V(H)$ satisfies: $\forall A, B \subseteq V(H)$ s.t. $2 \leq |A|, |B| \leq k$ and $A \cap M \neq \emptyset$, $B \cap M \neq \emptyset$ and $A \setminus M = B \setminus M \neq \emptyset$ then $A \in \mathcal{E}(H) \Leftrightarrow B \in \mathcal{E}(H)$.*

Definition 1.2.6. *(**Courcelle's module** [26]) Given a hypergraph H, we call **Courcelle's module** a subset $M \subseteq V(H)$ that satisfies $\forall A \in \mathcal{E}(H)$, $A \cap M = \emptyset$ or $A \setminus M = \emptyset$, or $M \setminus A = \emptyset$.*

In Chapter 2 we will see that each of these three different definitions of module leads to a unique decomposition theorem via the properities of partitive families [19].

We then developed a general algorithmic scheme following the idea of a work of modular decomposition for graphs [52], generalize it for standard hypergraph module and k-subset module. The algorithmic scheme assumes we know computing a function $Minmodule(\{x, y\})$, $\forall x, y \in V$, that is, the smallest module that contains vertices x and y, and builds the decomposition tree based on calls to $Minmodule(\{x, y\})$, $\forall x, y \in V$.

Theorem 1.2.1. *For every partitive family $\mathcal{F}$ over a ground set V, its decomposition tree can be computed using $O(|V|^2)$ calls to $Minmodule(\{x, y\})$, with $x, y \in V$.*

So if computing the function $Minmodule(\{x, y\})$ can be done in $O(p(n))$time, then the computation of the decomposition tree can be done in $O(n^2 \cdot p(n))$. We then showed that for standard hypergraph module and k-subset module, we can compute $Minmodule(\{x, y\})$ both in $O(n \cdot l)$ time, where l is l the sum of the size of the edges.

Lemma 1.2.1. *For a simple hypergraph H and $A \subsetneq V(H)$, there is an algorithm computes the minimal standard module that contains A in $O(n \cdot l)$ time.*

Lemma 1.2.2. *For a simple hypergraph H and $A \subsetneq V(H)$, there is an algorithm, s.t. for any input integer $k \leq |V(H)|$ it can compute the minimal k-subset module that contains A in $O(n \cdot l)$ time.*

Combine these results, we get $O(n^3 \cdot l)$ algorithms for modular decomposition of standard module and k-subset module.

Theorem 1.2.2. *For a simple hypergraph H, the modular decomposition for standard module and k-subset module could be computed in $O(n^3 \cdot l)$ time.*

1.2.5 Contribution

We studied the generalization of modular decomposition, including two for graphs and three variations of definition of modules for hypergraphs.

For ϵ-module in graphs, we showed that ϵ-module and ϵ-splitter module are not partitive and one-step parallel decomposition of ϵ-module with $\epsilon = 1$ is NP-hard.

For hypergraphs,

1. We found three variations of modules in the literature: the standard modules [73], the k-subset modules [13] and the Courcelle's modules [26], each one of them leads to a unique decomposition.

2. We developed a general algorithmic scheme following the idea in [52], to compute the decomposition tree of a partitive family on ground set V using $O(|V|^2)$ calls to $Minmodule(\{x, y\})$, with $x, y \in V$.

3. We proved that $Minmodule(\{x, y\})$ of standard module and k-subset module could be computed both in $O(n \cdot l)$ time, which result in $O(n^3 \cdot l)$ algorithm for modular decomposition of standard module and k-subset module, improving the previous result based on a $O(n^4 m^3)$ algorithm [74] and $O(n^{3k-5})$ algorithm [13] respectively, also conclude the decomposition of k-subset module in hypergraphs is in P.

1.3 Generalized Binary Search Problem

In large graph applications, it happens quite often that we are usually operating with distributively stored information, the access of datas is realized by querying to a storage that we call them oracles. One factor influencing the efficiency is the time of queries, for example, when we request for information from a remote server, there will be time delay for receiving the responses. Here we study search problem, a kind of modelization for information exploration.

1.3.1 Search Problem

The search problem is to locate a "hidden" target node in a graph by asking queries to a given oracle. Assume we have a graph $G = (V, E, w)$ with weight function $w \colon V \to \mathbb{R}_+$ and a target node x, each time we selects a vertex v in the graph, ask to the oracle the partial information of x related to vertex v, which we call a "query", and after time $w(v)$, we receive the information of x, and $w(v)$ is regarded as the cost of the query.

Many applications have the similar oracles that provide partial information, for example, locating an element in tree-organized data like XML, oracles may only reply "if the target is a child of node v?" for a given v. Also, in classification problem, where every element has some attributes, and we want to classify them into several classes, we are not able to know which class the element belongs to at a single step, but we perform tests as "is the first attribute of the element satisfies the v-th class?" and so on. More over, in these applications, the cost of query may not

be restricted to the real response time, but also could be the computation resource consumption, etc.

Depend on different query model, ways to design search strategies are different. We present firstly the formalization of search strategies, then our generalized binary search model.

Search strategy. A **search strategy** $\mathcal{A}$ for a graph $G = (V, E, w)$ is an adaptive algorithm which defines successive queries to the graph, based on responses to previous queries, with the objective of locating the target vertex in a finite number of steps.

A search strategy could be described by $\mathtt{Q}_{\mathcal{A}}(G, x)$ the time-ordering (sequence) of queries performed by strategy $\mathcal{A}$ on graph G to find a target vertex x, with $\mathtt{Q}_{\mathcal{A},i}(G, x)$ denoting the i-th queried vertex in this time ordering, $1 \leq i \leq |\mathtt{Q}_{\mathcal{A}}(G, x)|$.

Cost of a strategy. We denote by $\mathtt{COST}_{\mathcal{A}}(T, x) = \sum_{i=1}^{|\mathtt{Q}_{\mathcal{A}}(T,x)|} w(\mathtt{Q}_{\mathcal{A},i}(T, x))$ the sum of weights of all vertices queried by $\mathcal{A}$ with x being the target node, i.e., the time after which $\mathcal{A}$ finishes. Let

$$\mathtt{COST}_{\mathcal{A}}(T) = \max_{x \in V} \mathtt{COST}_{\mathcal{A}}(T, x)$$

be the **cost** of $\mathcal{A}$. We define the *cost of* T to be

$$\mathtt{OPT}(T) = \min\{\mathtt{COST}_{\mathcal{A}}(T) \;\big|\; \mathcal{A} \text{ is a search strategy for } T\}.$$

The goal is to design a search strategy that locates the target node and minimizes the search time in the worst case. We say that a search strategy is **optimal** for T if its cost equals $\mathtt{OPT}(T)$. For given T, we say that a search strategy $\mathcal{A}$ is a $(1+\varepsilon)$-**approximation** of the optimal solution if $\mathtt{COST}_{\mathcal{A}}(T) \leq (1+\varepsilon)\mathtt{OPT}(T)$.

1.3.2 Generalized Binary Search

The generalized binary search model, as its name suggests, is generalized from the binary search: considering searching for an element in a sorted array, this could be seen as a problem of searching for a target node in a path, each query selects a node, and the oracle replies which 'side' (or sub-path) of the queried node the target node belongs to. When we generalize the structure of path to graphs, this coincides with the generalized binary search model. We give a precise description here.

Generalized Binary Search model. In generalized binary search model, each **query** selects a node v in the graph and after the time $w(v)$, the oracle gives a **reply**: the reply is either *true* which implies that v is the target node and thus the search terminates or it returns a neighbor u of v which lies closer to the target x than v (equivalent to a neighbor that belongs to the shortest path between x and v). In a general graph, this node may not be unique and we assume that the oracle may reply any one of these nodes. If the graph is a tree, such a neighbor u

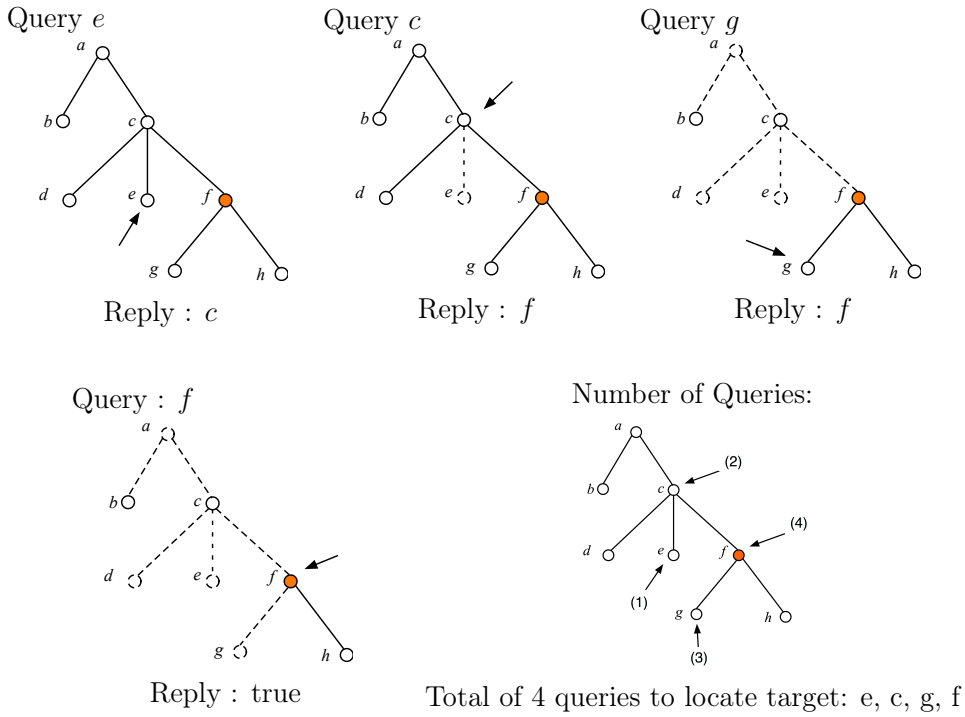


Figure 1.3: An example of searching a target node (assumed f) in a tree with a generalized binary search query model.

is unique and is equivalently described as the unique neighbor of v belonging to the same connected component of $T \setminus \{v\}$ as x.

By tuning several settings, there are variations of generalized binary search model, we list here the main variations: reliable/unreliable oracle, edge/mode query oracle.

Variation: reliable/unreliable oracles. The oracle is reliable if it always replies the correct information to the query, but there are also studies on oracles that doesn't always reply correct answers. The generalized binary search with an oracle whose answer to a query is correct with some probability $p > \frac{1}{2}$ is studied in [8, 37, 38, 53]. And a model in which a fixed number of queries can be answered incorrectly during a binary search is studied in [79]. The model with an adversarial error rate bounded by a constant $r \leq 1/2$ is studied in [32].

Variation: edge/vertex queries. We could also consider the version that each query is on an edge, and the oracle reply one of the incident vertex that is closer to the target. In fact, the edge-query variant can be reduced to vertex-query model by first assigning a 'large' weight to each vertex of G (for example, one plus the sum of the weights of all edges in the graph) and then subdivide each edge e of G giving to the new node the weight of the original edge, $w(e)$. So the vertex-query model is

more general than the edge-query model. However, in history, the edge variant of trees has been more studied [60, 72, 29, 28, 24, 23].

In the following of this thesis, when we say "search problem", we assume that we are restricting to the search problem with classical (reliable, node-query) generalized binary search model, except otherwise stated.

Different graph types. There are also many studies on different type of graphs on edge or vertex query model. From a structural aspect, there are mainly on paths [23], trees [76, 81, 31], directed or undirected general graphs [37] and partial orders [17, 29, 76]. From an quantitive aspect, there is difference between weighted or unweighted graph (meaning the uniform or non-uniform query costs), the difference is discussed in some of the works listed, and we will present it more detailed nextly in Section 1.3.3.

1.3.3 Related Works

In this thesis, we study the generalized binary search with reliable oracles and vertex-queries. As the existence of reduction of edge-query to vertex-query, we don't need to state for edge-query model and the results for vertex-query naturally apply for edge-query model.

We list here the complexity results for different type of graphs.

Table 1.1: Computational complexity of the search problem in different graph classes, including our results for weighted trees. Completeness results refer to the decision version of the problem.

Graph class	*Unweighted*	*Weighted*
Paths:	exact in $O(n)$ time	exact in $O(n^2)$ time [23]
Trees:	exact in $O(n)$ time [76, 81]	strongly NP-complete [31]
Undirected:	exact in $n^{O(\log n)}$ time [37]	PSPACE-complete [37]
	$O(\log n)$-approx. in poly-time [37]	$O(\log n)$-approx. in poly-time [37]
Directed:	PSPACE-complete [37]	PSPACE-complete [37]

An optimal search strategy can be computed in linear-time for an unweighted tree [76, 81]. The number of queries performed in the worst case may vary from being constant (for a star one query is enough) to being at most $\log_2 n$ for any tree [76] (by always querying a node that halves the search space). Several following results have been obtained in [37]. First, it turns out that $\log_2 n$ queries are always sufficient for general simple graphs and this implies a $O(m^{\log_2 n} n^2 \log n)$-time optimal algorithm for arbitrary graphs. The algorithm which performs $\log_2 n$ queries also serves as a $O(\log n)$-approximation algorithm, also for the weighted version of the

problem. On the other hand, it has been proven that optimal algorithm with a running time of $O(n^{o(\log n)})$ is in contradiction with the Exponential-Time-Hypothesis, and for $\varepsilon > 0$, $O(m^{(1-\varepsilon)\log n})$ is in contradiction with the Strong Exponential-Time-Hypothesis. When non-uniform query times are considered, the problem becomes PSPACE-complete. Also, a generalization to directed graphs also turns out to be PSPACE-complete.

1.3.4 Outline of Our Work

We study the generalized binary search problem in weighted trees. Given a node-weighted rooted tree $T = (V, E, w)$ with weight function $w\colon V \to \mathbb{R}_+$, we would like to have good strategies to find the unknown target node x with generalized binary search oracle. As designing an optimal strategy for a weighted tree search instance is known to be strongly NP-hard [31], we aim at computing an approximation of the optimal strategy.

In order to apply classical approximation techniques to this problem, following the idea in [28], we model in Section 3.3.2 any search strategy as a **consistent schedule**, where each node is associated with a job that has a fixed processing time set to the weight of node, and jobs need to satisfy the **consistent** constraints. We then see that we have a description similar to scheduling problems, but with non-classical constraints. With this model, we are able to apply techniques in approximation algorithm for scheduling problems as rounding [88].

In Section 3.4.1 and 3.4.2, we apply rounding techniques to both the cost function and starting times of the jobs in a schedule, and finally created the **aligned schedule**, such that any optimal strategy could be turned into an aligned consistent schedule whose modified cost function is a $(1+\varepsilon)$-approximation of the optimal strategy. Then if we could enumerate all aligned consistent schedules, we are able to obtain the optimal aligned schedule, and a $(1+\varepsilon)$-approximation of cost of the optimal strategy.

In section 3.4.3, we propose a dynamic programming algorithm to enumerate aligned consistent schedules, and runs in quasi-polynomial time. Then in section 3.4.4, we explain how to extract the search strategy from the optimal aligned consistent schedule, together with 3.4.5 this procedure adds no more than $\varepsilon\mathtt{OPT}(T)$ to the cost, so the obtained search strategy is a $(1+\varepsilon)$-approximation of the optimal, and our algorithm computing the strategy runs in quasi-polynomial time, thus the QPTAS of the generalized binary search problem in weighted trees, and we get our first theorem on generalized binary search problem in weighted trees:

Theorem 1.3.1. *There exists an algorithm running in $n^{O\left(\frac{\log n}{\varepsilon^2}\right)}$ time, providing a $(1+\varepsilon)$-approximation solution to the generalized binary search problem in weighted trees.*

Based on our QPTAS, we could also build a polynomial time approximation algorithm. In Section 3.5, following the general idea from [24], we apply a recursively partition method, whose objective is to partition the tree into small subtrees, such that the sizes of these subtrees are small enough then we could solve the problem

on these subtrees with our QPTAS in polynomial time with respect to $|T|$, so combining solutions of these subtrees, we get a $O(\sqrt{\log n})$-approximation algorithm in polynomial time:

Theorem 1.3.2. *There is a $O(\sqrt{\log n})$-approximation polynomial time algorithm for the generalized binary search problem in weighted trees.*

1.3.5 Contribution

We work on the generalized binary search problem in weighted trees, show that:

1. The problem admits a quasi-polynomial time approximation scheme: for any $\varepsilon > 0$, there exists a $(1+\varepsilon)$-approximation strategy with a computation time of $n^{O(\log n/\varepsilon^2)}$. Thus, the problem is not APX-hard, unless $NP \subseteq DTIME(n^{O(\log n)})$.

2. By applying a generic reduction, we obtain as a corollary that the studied problem admits a polynomial-time $O(\sqrt{\log n})$-approximation. This improves previous $\hat{O}(\log n)$-approximation approaches, where the $\hat{O}$-notation disregards $O(\text{poly}\log\log n)$-factors.

1.4 Pure-LOCAL Model and Max-flow Problem

Here we present the model we work on in distributed computing environment, and build the approximation algorithm for maximum $s-t$ flow problem on it.

We first introduce the basic settings in distributed computing and present our model.

Distributed Computing In the classical configuration of distributed computing, machines are organized into a graph $G = (V, E)$, where each vertex (or node) has an unique identifier and represents a machine that has computation ability and resources, and each edge $(u, v) \in E$ represents a communication link between machines u, v, meaning u and v could send messages (a number of bits) to each other directly. We could assume here that the graph is undirected, connected and simple. We call u, v are neighbors if $(u, v) \in E$, and denote $N(u)$ as all the neighbors of $u \in V$.

Every vertex and edge could be associated with a **label**, the state of the labels of all vertices and edges constitutes a **configuration**. In distributed computing, an **input** is a configuration set to the graph, and an **output** is the configuration after some operations (or the execution of an algorithm) on the graph.

An algorithm here is called a distributed algorithm, compared to classical algorithms, here we could have two extra operations, **send** and **receive** messages. A send operation could send messages to its selected neighbors (could be any one, some or all). A receive operation, symmetrically, could receive messages from its selected neighbors, but only receive in success if the selected neighbor had sent to it.

In the study of distributed algorithms, being different than some other disciplines of distributed computing, we are not focusing on the synchronization problem. Here

we assume all the machines are synchronized with **rounds**, in a round, each machine can receive messages from its neighbors, do its local computation, and send messages to its neighbors.

The complexity measure of a distributed algorithm is thus the number of rounds from getting the input to get the output. And depend on different models, there might be other measure or constraints on computing resources or graph structures, in the next section we present our model.

1.4.1 Pure-LOCAL Model

We consider a model with limited memory per node, specifically every node can only store a constant number of variables per degree:

Pure-LOCAL. We define our Pure-LOCAL model as following: (i) The network is represented as a graph $G = (V, E)$, where edges represent the bidirectional communication link, and each node only knows and can communicate with its neighbors. (ii) Communications are synchronized, this means each node can proceed one receive and one send operation from/to each neighbor during one round, and we measure the time complexity by number of rounds. (iii) A node can only store and one send operation could only send a constant number of variables for each of its neighbors.

If the degree of a vertex u is $deg(u)$, then it implies the node could store $O(deg(u))$ variables. Here if we assume real variables are stored in constant number of bits, it implies a limited size of memory of $O(\Delta \log n)$ bits per node and $O(\log n)$ message size, where Δ is the maximum degree of all nodes in the graph.

The motivation to study this model is to impose the memory efficiency and locality of computation. The classical LOCAL model [63, 77] does not bound computing power per node and the CONGEST model [77] put a restriction on LOCAL model with $O(\log n)$ bits of message size, but no constraints on computing resource per node. These two classical model both admit a general framework of gathering all the information on one node and launch computations on this node, and this kind of framework is applied in a lot of scenes of algorithm design on LOCAL/CONGEST model.

Although the two models support well the study on locality and in situations that the computing power of a single machine is not an important concern than communication costs, the assumption of unbounded computing power on machines is less realistic in some scenarios. For example, in the network of low energy devices, like wearable or medical devices, smart domestic systems, the execution of complex or high complexity algorithms could be difficult. Another example is the natural algorithms, which studies the algorithms on biological objects, as animals, microbes and chemicals, it seems less natural to assume their behaviors are following complex algorithmic rules.

Our model could be seen as a classical CONGEST model with a limit of memory size on each node, and naturally being more constrained than CONGEST model. This is a memory-efficient assumption but also reflects the requirement of simplicity of algorithm on each machine.

In our work, we study the algorithm design for maximum $s-t$ flow problem on this model.

1.4.2 Maximum $s-t$ Flow Problem

The maximum $s-t$ flow problem is well known in combinatorial optimization [82, 1] and has many application in logistics, transportation and industry engineering [2], and its dual problem, the minimum $s-t$ cut, has also a application of the famous graph cut method [16] which receives recent attentions in image segmentations, and is implemented by solving the maximum $s-t$ flow problem.

The problem could be stated as following:

Let $G=(V,E)$ be an undirected graph, with n vertices and m edges, among which there are two special vertices, a source s and a sink t. Every edge is assigned with a positive integral capacity $U_e \in \mathbb{Z}^+$. Let A be the adjacency matrix of G

Definition 1.4.1. *An s-t flow is a function $f: E \to \mathbb{R}$ obeying the flow-conservation constraints*

$$\sum_{e \in N(v)} f(e) = 0, \text{ for all } v \in V \setminus \{s,t\}$$

The value of flow $|f|$ is defined by $|f| := \sum_{e \in N(s)} f(e)$ and is equal to $\sum_{e \in N(t)} f(e)$ by flow-conservation.

Here U_e is the edge capacity and we assume that is polynomial w.r.t. n. An s-t flow is feasible for capacities U_e if $\forall e \in E$, $|f(e)| \leq U_e$. The maximum s-t flow problem is to find a feasible s-t flow with maximum value.

In centralized settings, many polynomial combinatorial algorithms have been proposed to solve maximum $s-t$ flow problem in history, including the Ford-Fulkerson algorithm [39], Edmonds-Karp algorithm [36], push-relabel algorithm [44], etc. And recently, another series of study is raised on using continuous optimization method to compute $(1+\varepsilon)$ approximation of maximum $s-t$ flow [20, 61, 68] their basic framework is to regard the maximum $s-t$ flow problem as an optimization problem defined by a Laplacian system (which is equivalent to the description of the electrical flow in circuits) and optimize with a Laplacian solver [55], where two main optimization techniques are adopted to approximate the optimal solution: the multiplicative weights update method [4, 20] and gradient descent based method [61]. Our work is based on one of these works [20] based on multiplicative weights update method and electrical flow and we will present basics in later sections.

Solving this problem in classical distributed configurations has also been studied, and algorithms are basically designed by implementing one or a combination of classical centralized algorithms to an distributed version. The push-relabel algorithm [44] could be naturally implemented in Pure-LOCAL model and runs in $O(V^2E)$ time. In [42] authors proposed an $(1+o(1))$-approximation algorithm on CONGEST model in $(D+\sqrt{n})n^{o(1)}$ rounds, which their method depend on gradient descent method and hard to implement in Pure-LOCAL model.

Although the push-relabel algorithm could fits the local computation, we are now interested in the question that whether non-combinatorial algorithms for Max-flow problem could be implemented in Pure-LOCAL model. And we study the algorithm of [20] by Christiano, Kelner, Mądry, Spielman and Teng. The reason to study this category of algorithms in Pure-LOCAL model is that the general optimization techniques they use could be applied to many other problems, for example, the work on Physarum dynamics [5] also suggests a similar procedure combining electrical flow and re-weighting, and we wish that our method could also be applied to implementing these procedures in Pure-LOCAL model.

1.4.3 Approximating Max-flow by Multiplicative Weights Update

An algorithm approximating Max-flow by solving electrical flow with weights updating is presented in [20]. Intuitively, one can think the setting of Max-flow is quite similar to electrical flow, but with a different objective function. Can we adjust resistances in an electrical circuit such that the electrical flow tends to the max-flow in this graph?

In [20], Mądry et al utilize the electrical flow connection and iteratively choose resistances in an constant current source electrical network, such that the average of electrical flow they get among all rounds tends to the max-flow if the source flow value is close to the max-flow value. In fact they are considering the decision problem of approximating maximum $s-t$ flow problem.

Decision problem of approximating Max-flow problem. The decision problem of $(1+\varepsilon)$ approximation of Max-flow is that, given a graph $G=(V,E)$ with edge capacity U_e, $e \in E$ and a value F, we'd like to check if F exceed the $(1+O(\varepsilon))$ of the maximum flow and otherwise returns an $s-t$ flow $\bar{f}$ such that $|\bar{f}| \geq (1-O(\varepsilon))F$ and for every $e \in E$, $\bar{f}_e \leq U_e$.

If there is an algorithm solves the $(1+\varepsilon)$ decision problem of Max-flow, we could compute a $(1+O(\varepsilon))$ approximation of the Max-flow by launching a binary search on value F, i.e. set $F=1$ and run the algorithm, then multiply F by $(1+\varepsilon)$ each time until there returned an "no". Then the previous value is a $(1+O(\varepsilon))$ approximation, and this binary search procedure brings an $\log_{(1+\varepsilon)} F_{max}$ factor to the running time.

Multiplicative weight update method for Max-flow. In [20], Mądry et al proposed the following weight-updating method for a given test-value F:

$$
\begin{aligned}
w_e^{(0)} &= 1, \forall e \in E \\
r_e^{(t)} &= \frac{1}{U_e^2}(w_e^{(t)} + \frac{\varepsilon \left\|\boldsymbol{w}^{(t)}\right\|_1}{3m}) \\
f_e^{(t)} &= \text{electrical flow induced by } \boldsymbol{r}^{(t)} \text{ and } F \\
cong(f_e^{(t)}) &= \frac{f_e^{(t)}}{U_e} \\
w_e^{(t)} &= \left(w_e^{(t-1)}\right)\left(1 + \frac{\varepsilon}{\rho} cong_e(f^{(t-1)})\right)
\end{aligned}
\tag{1.1}
$$

Where U_e is the edge capacity and $cong_e(f) = \frac{f_e}{U_e}$ is the **congestion** of an edge, ρ is a parameter. They showed if F is a $(1 + O(\varepsilon))$ approximation of Max-flow, then an $(1 - \varepsilon)$ factor of the average of these electrical flow computed in all rounds satisfy the edge capacity constraints and is an $(1 + O(\varepsilon))$ approximation of Max-flow, and can return "NO" if F not. In fact, they also show that the computation of the electrical flow does not need to be exactly, i.e. an approximate solution of the electrical network is enough.

Challenges of implementation in Pure-LOCAL model. To implement the Algorithm 1.1 in Pure-LOCAL model, there are two main challenges:

1. How to compute $\frac{\left\|\boldsymbol{w}^{(t)}\right\|_1}{m}$ the average of weights ? It's not trivial because $\left\|\boldsymbol{w}^{(t)}\right\|_1$ is a global quantity and is changing between iterations.

2. How to compute the electrical flow induced by $\boldsymbol{r}^{(t)}$ and F ? Computing an electrical flow could be reduced to solve a Laplacian system as we will present in Section 4.2.1, but there is no known general Laplacian solver in Pure-LOCAL model with weight-updating.

1.4.4 Outline of Our Work

We study the design of algorithm in [20], by Mądry et al in Pure-LOCAL model.

As in the above section, we have two challenges in turning Mądry et al's algorithm in Pure-LOCAL model: estimating the average of weights and approximating the electrical flow. We propose to solve these two challenges by matrix computations from the idea of random walk.

We are going to show that, the **simple random walk in a graph** could approximate the average of weights, and the **weighted random walk with edge conductances** could approximate the flow of an electrical network. To handle the issue that weights are changing with iterations, we design the "slow down" parameters, such that the weights do not change too much before we get an approximation that has an error small enough of the quantities we need.

In Section 4.2.2 we present the connection of Max-flow problem and electrical flow and the idea of approximating Max-flow by optimization technique and solving electrical networks. In Section 4.2.1 we present the preliminary connection of electrical flow and graph Laplacian that compute an electrical flow could be reduced to

solve a Laplacian system. In Section 4.2.3 we introduce the basics of random walk, and the idea to approximate intended quantities in our algorithm.

In Section 4.3 we describe our algorithm both in dynamical system way and Pure-LOCAL distributed algorithm way. In Section 4.4.2 and Section 4.4.1 we show that the re-weighting process in our algorithm provide a vector approximating $(1+O(\varepsilon))$ Max-flow if there could be an $(1+\varepsilon)$-energe approximation of electrical flow. In Section 4.4.3 we show that our algorithm build an $(1+\varepsilon)$-energe approximation of electrical flow.

We need to precise that, our approximation algorithm in Pure-LOCAL model does not provide a solution to the original decision problem but a weaker version:

Weaker decision problem of Max-flow approximation. Our algorithm solve a weaker decision problem of $(1+\varepsilon)$ approximation of Max-flow: given a graph $G=(V,E)$ with edge capacity U_e, $e \in E$ and a value F, if F does not exceed the maximum flow, we will return "YES" . And if we return "NO" then F must exceed the maximum flow. More precisely, let F^* be the max-flow value, we will compute a vector $\bar{\boldsymbol{f}}$, such that if $F \leq F^*$ then $cong_e(\bar{\boldsymbol{f}}) \leq 1$, and if $cong_e(\bar{\boldsymbol{f}}) > \frac{(1+\varepsilon)^2}{(1-\varepsilon)^2}$ then $F > F^*$.

This weakness compared to the centralized algorithm in [20], is that the approximated electrical flow that our algorithm produces may have an error on flow-conservation constraints, thus we are not able to ensure the vector $\bar{\boldsymbol{f}}$ is an $s-t$ flow and lead to this one-side result.

1.4.5 Contribution

We studied the implementation of an multiplicative weights update algorithm approximating Max-flow problem in Pure-LOCAL model.

1. We design the way in Pure-LOCAL model to approximate the global quantity of average weight in a weight-varying iterative algorithm.

2. We implement in Pure-LOCAL model the computation to solving an electrical network with resistances changing in iterations. As we know, this is the first algorithm approximating the electrical flow in a weight-varying environment.

3. We show that our algorithm solves a weaker version of Max-flow approximation decision problem in polynomial time.

Perspective We'd like to apply our method to implement other weight-update based algorithm in Pure-LOCAL model, i.e. the Physarum dynamics [5] that computes the shortest path, etc.

1.5 Publications during PhD

- Michel Habib, Fabien de Montgolfier, Lalla Mouatadid, Mengchuan Zou:
  A General Algorithmic Scheme for Modular Decompositions of Hypergraphs and Applications.
  International Workshop on Combinatorial Algorithms (IWOCA) 2019, invited to a special issue of Journal *Theory of Computing Systems*

- Michel Habib, Lalla Mouatadid, Mengchuan Zou:
  Approximating Modular Decomposition is Hard.
  Accepted to The International Conference on Algorithms and Discrete Applied Mathematics (CALDAM) 2020.

The second publication here presents also some positive results of enumerating all minimal ε-modules, which are not included in this thesis.

- Dariusz Dereniowski, Adrian Kosowski, Przemyslaw Uznanski, Mengchuan Zou:
  Approximation Strategies for Generalized Binary Search in Weighted Trees.
  International Colloquium on Automata, Languages, and Programming (ICALP), 2017

Chapter 2

Generalization of Modular Decompositions

This chapter considers the generalization of modular decomposition and its efficient algorithms. We study two type of generalizations: (1)Modular decomposition in hypergraphs; (2)Allowing errors in modules of graphs. We present both positive and negative (hardness) results.

We first present the hypergraph modular decomposition. In the literature we find three different definitions of modules, namely: the standard one [73], the k-subset modules [13] and the Courcelle's one [26]. They all lead to partitive families, and thus each accepts an unique decomposition tree. For Courcelle's module an linear-time decomposition algorithm is already known [18], and we study designing efficient algorithms for the standard and the k-subset modular decomposition of hypergraphs.

When allowing errors in graph modules, we look at two definitions: ϵ-module and ϵ-splitter module, we conclude that they are not partitive thus do not satisfy the condition of the unique modular decomposition theorem. More over, testing of one-step parallel decomposition ϵ-module for $\epsilon = 1$ is already NP-hard.

2.1 Preliminaries and Definitions

2.1.1 Classical Modular Decomposition

Let G be a simple, loop-free, undirected graph, with vertex set $V(G)$ and edge set $E(G)$, $n = |V(G)|$ and $m = |E(G)|$ are the number of vertices and edges of G respectively. $N(v)$ denotes the neighbourhood of v and $\overline{N(v)}$ the non-neighbourhood, this notation could also be generalized to set of vertices, i.e. $N(X)$ (resp. $\overline{N(X)}$) , for $X \subseteq V(G)$, are vertices who have (resp. haven't) a neighbour in X.

Definition 2.1.1. *For an undirected graph G, $M \subseteq V(G)$ is a **module** if and only if: $\forall x, y \in M$, $N(x) \setminus M = N(y) \setminus M$.*

In other words, $V(G) \setminus M$ is partitioned into X, Y such that there is a complete bipartite between M and X, and no edge between M and Y. For convenience let

us denote X (resp. Y) by $N(M)$ (resp. $\overline{N(M)}$). For $x, y \in V$, we call them **false-twins** if $N(x) = N(y)$ and **true-twins** if $N(x) \cup \{x\} = N(y) \cup \{y\}$. It's easy to see that all vertices within a module are at least false twins.

Here we give an example of a graph with a partition by strong modules. [1]

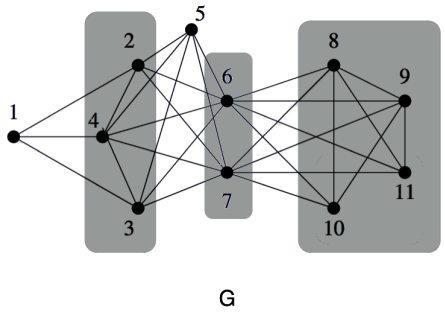


Figure 2.1: Strong modules in G are: {1}, {2,3,4},{5},{6,7}, {8,9,10,11}

A single vertex $\{v\}$ and V are always modules, and called **trivial modules**. A graph that only has trivial modules as induced subgraphs is called a **prime graph**. Two non-empty sets A and B **overlap** if $A \cap B \neq \emptyset, A \setminus B \neq \emptyset$, and $B \setminus A \neq \emptyset$. A **strong module** is a module that does not overlap with other modules.

Definition 2.1.2. *A **modular decomposition tree** is defined as follows [13]:*

(a) tree nodes are strong modules;

(b) parent relation is the containment relation of tree nodes;

*(c) each internal node with only two children is labeled as **complete** and each other internal node is labeled as*

- ***complete** if the union of any subset of its children is a module;*
- ***prime** if each of its children is a module while no other union of a proper subset of its children is a module.*

In the case of graphs, in fact *complete* nodes could also be distinguished by two types of operations: *parallel* (disjoint union) and *series* (connect every pair of nodes in disjoint sets X and Y).

By the Modular Decomposition Theorem [41], any graph accepts a unique **modular decomposition tree**. A graph is **totally decomposable** if there is no prime node in the decomposition tree. Totally decomposable graphs with respect to modular decomposition are also known as cographs in the literatture, or P_4-free graphs.

[1] This example is taken from the teaching materials of Michel HABIB.

Cographs form a well studied graph class where many classical NP-hard problems such as maximum clique, maximum independent set, Hamiltonicity become tractable, see for instance [25].

Sets that do not overlap are said to be **orthogonal**, which is denoted by $A \perp B$. Let $\mathcal{F}$ be a family of subsets of a ground set V. A set $S \in \mathcal{F}$ is called **strong** if $\forall S' \neq S \in \mathcal{F} : \mathcal{S} \perp \mathcal{S}'$.

Definition 2.1.3. *[19] A family of subsets $\mathcal{F}$ over a ground set V is **partitive** if it satisfies the following properties:*

(i) *$\emptyset$, V and all singletons $\{x\}$ for $x \in V$ belong to $\mathcal{F}$.*

(ii) *$\forall A, B \in \mathcal{F}$ that overlap, $A \cap B, A \cup B, A \setminus B$ and $A \Delta B \in \mathcal{F}$, here Δ denote the symmetric difference operation.*

The study on partitive families extends the results of [41], in [19] they show every partitive family admits a unique decomposition tree, with the two types of nodes as in Definition 2.1.2. In this decomposition tree, every node corresponds to a set of the elements of the ground set V of $\mathcal{F}$, and the leaves of the tree are single elements of V, strong elements of $\mathcal{F}$ form a tree ordered by the inclusion relation. For a complete (resp. prime) node, every union of its child nodes (res. no union of its child nodes other than itself) belongs to the partitive family.

The uniqueness of decomposition tree for partitive families provides us the foundation to study the generalization of modular decomposition upon families that are partitive.

2.1.2 Hypergraphs

Following Berge's definition of hypergraphs [9], a hypergraph H over a finite ground set $V(H)$ is made by a family of subsets of $V(H)$, denoted by $\mathcal{E}(H)$ such that (i) $\forall e \in \mathcal{E}(H)$, $e \neq \emptyset$ and (ii) $\cup_{e \in \mathcal{E}(H)} e = V(H)$. We assume here also a hypergraph admits no empty edge and no isolated vertex.

When analyzing algorithms, we use the standard notations: $|V(H)| = n$, $|\mathcal{E}(H)| = m$ and $l = \Sigma_{e \in \mathcal{E}(H)} |e|$. For every edge $e \in \mathcal{E}(H)$, we denote by $H(e) = \{x \in V(H)$ such that $x \in e\}$, and for every vertex $x \in V(H)$, we denote by $N(x) = \{e \in \mathcal{E}(H)$ such that $x \in H(e)\}$.

To each hypergraph one can associate a bipartite graph G, namely its incidence bipartite graph, such that: $V(G) = V(H) \cup \mathcal{E}(H)$ and $E(G) = \{xe$ with $x \in V(H)$ and $e \in \mathcal{E}(H)$ such that $x \in H(e)\}$.

A hypergraph is **simple** if all its edges are different. In this case $\mathcal{E}(H) \subseteq 2^{V(H)}$.

For a hypergraph H and a subset $M \subseteq V(H)$, let $H(M)$ denote the *hypergraph induced by* M, where $V(H(M)) = M$ and $\mathcal{E}_{H(M)} = \{e \cap M \in \mathcal{E}(H)$, for $e \cap M \neq \emptyset\}$. Similarly, let H_M denote the *reduced hypergraph* where $V(H_M) = (V \setminus M) \cup \{m\})$ with $m \notin V$, and $\mathcal{E}(H_M) = \{e \in \mathcal{E}(H)$ with $e \cap M = \emptyset\} \cup \{(e \setminus M) \cup \{m\}$ with $e \in \mathcal{E}(H)$ and $e \cap M \neq \emptyset\}$. By convention in case of multiple occurences of a similar edge, only one edge is kept and so H_M is a simple hypergraph.

2.2 Variant Definitions of Hypergraph Modules

2.2.1 Standard Modules

In the literature three variations of modules defined in the hypergraphs, and we first introduce the one defined in [73] that we called a standard module.

Definition 2.2.1 (Hypergraph Module)**.** *Given a hypergraph H, a* ***standard module*** $M \subseteq V(H)$ *satisfies:* $\forall A, B \in \mathcal{E}(H)$ *s.t.* $A \cap M \neq \emptyset$, $B \cap M \neq \emptyset$ *then* $(A \setminus M) \cup (B \cap M) \in \mathcal{E}(H)$.

The reason we name it as standard is that this definition relates to the following hypergraph substitution operation.

Hypergraph substitution: *Substitution* in general is the action of replacing a vertex v in a graph G by a graph $H(V', E')$ while preserving the same nerighbourhood properties. To apply this concept to hypergraphs, we use the definition presented in [74, 73]:

Definition 2.2.2. *Given two hypergraphs H, H_1, and a vertex $v \in V(H)$ we can define another hypergraph H' obtained by substituting in H the vertex v by the hypergraph H_1, and denoted by $H' = H_v^{H_1}$ which satisfies $V(H') = \{V \setminus v\} \cup V(H_1)$, and $\mathcal{E}(H') = \{e \in \mathcal{E}(H)$ s.t. $v \notin e\} \cup \{f \setminus v \cup e_1$ s.t. $f \in \mathcal{E}(H)$ and $v \in f$ and $e_1 \in \mathcal{E}(H_1)\}$.*

Note that even if H, H_1 are undirected graphs, the substitution operation may create edges of size 3, and therefore the resulting hypergraph H' is no longer a graph.

Proposition 2.2.1. *The class of simple hypergraphs is closed under substitution.*

When M is a module of H then $H = (H_M)_m^{H(M)}$. On the previous example: let us take $A, B \in \mathcal{E}$ (resp. 1^{st} and 6^{th} columns of H') then $(A \setminus M) \cup B \cap M = 2^{nd}$ column of H' and therefore belongs to $\mathcal{E}$.

Let's consider the example below where hypergraphs are described using their incidence matrices. In this example, we substitute vertex v_3 in H by the hypergraph H_1 to create H':

If M is a module of H then $\forall e \in \mathcal{E}(H_M)$, the edges of H that strictly contain e and are not included in M are the same. In other words, *all edges in $\mathcal{E}(H_M)$ behave the same with respect to the outside*, which is an equivalence relation between edges.

Proposition 2.2.2. *[74] The family of all standard modules of a simple hypergraph H yields a partitive family on $|V(H)|$.*

Proof. Of course every singleton of $V(H)$ is a module and $V(H)$ itself is also a module.

Let us consider two modules A, B that overlap. Using the above definition via an equivalence relation between edges, it is clear that $A \cap B$ is also a module. Similarly as there exists at least one edge in $\mathcal{E}_{\mathcal{A} \cap \mathcal{B}}$ since H is simple, by transitivity of this relation $A \cup B$ is also a module.

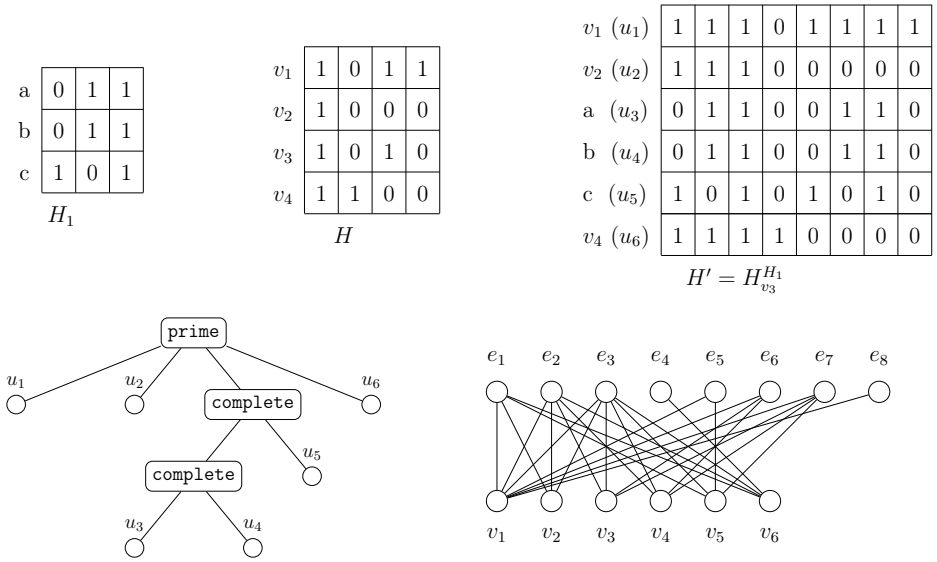


Figure 2.2: An example of substitution, its decomposition tree, and its incidence bipartite graph. $\{u_3, u_4, u_5\}$ is a module, but only u_3, u_4 are false twins in the incidence bipartite.

$A \setminus B$ the only case to check if the following: $F, F' \in \mathcal{E}(H(A \setminus B))$ such that $F \cup C \in \mathcal{E}(H)$ with $C \subseteq A \cap B$. Since B is an module, necessarily $F' \cup C \in \mathcal{E}(H)$. Therefore $A \setminus B$ is also a module of H.

Similarly, let $F \in \mathcal{E}(H(A \setminus B))$ such that $F \cup C \in \mathcal{E}(H)$ with $C \subseteq A \cap B$. Now if we consider $F' \in \mathcal{E}(H(B \setminus A))$, using the fact that B is a module and $C, F' \in \mathcal{E}(H(B))$ then $F \cup F' \in \mathcal{E}(H)$. But then since A is a module and $F, C \in \mathcal{E}(H(A))$ necessarily $C \cup F' \in \mathcal{E}(H)$. Thus, $A \Delta B$ is a module of H. □

Remark: if we use the following variant definition for hypergraphs :

Given a hypergraph H, a module $M \subseteq V(H)$ satisfies: $\forall A, B \in \mathcal{E}(H)$ s.t. $A \setminus M \neq \emptyset$, $A \cap M \neq \emptyset$, $B \setminus M \neq \emptyset$, $B \cap M \neq \emptyset$ then $(A \setminus M) \cup (B \cap M) \in \mathcal{E}(H)$.

Unfortunately this definition does not lead to a partitive family. Simply because using this definition, any set of size $|V|-1$ is a module, because any edge overlapping outside connects the same vertex. Then any set of size $|V| - 2$ is a module because it is the intersection of two sets of size $|V| - 1$. By induction we could have that any set is a module if the definition is partitive. However it could not be true. So the intersection property may fail.

Since every partitive family has a unique decomposition tree [19], it follows that the family of the modules of a simple hypergraph admits a uniqueness decomposition theorem and a unique hypermodular decomposition tree.

Consider the graph in Figure 2.3 which represents the incidence bipartite of the hypergraph H' constructed in the previous example, together with a renumbering of the vertices. As one can see, $\{v_3, v_4, v_5\}$ is a module. But only v_3, v_4 are false

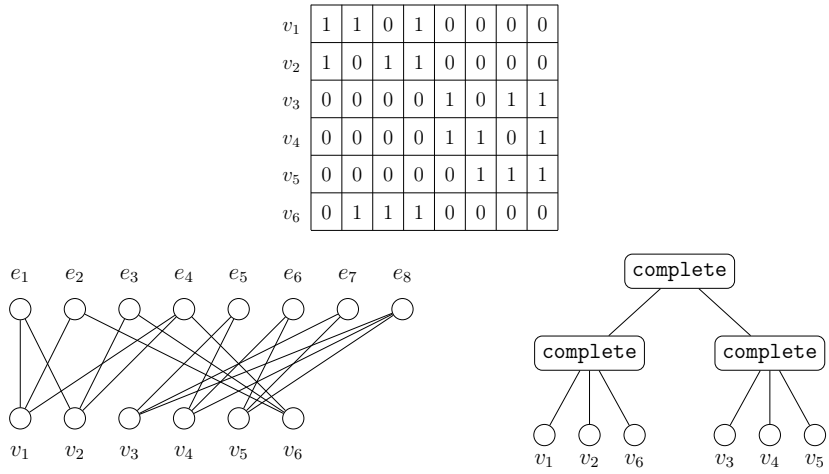


Figure 2.3: A hypergraph H given by its incidence bipartite and its modular decomposition tree

twins in the associated bipartite graph.

Modular decomposition of bipartite graphs just leads to the computation of sets of *false twins* in the bipartite graphs. So hypergraph modules are not always set of twins of the associated incidence bipartite.

Some authors [10, 11] defined **clutters** hypergraphs, in which no edge is included into another one. In this case, clutters modules are called **committees** [10]. Trivial clutters are closed under hypergraph substitution. The committees of a simple clutter also yields a partitive family which implies a uniqueness decomposition theorem. From this one can recover a well-known Shapley's theorem on the modular decomposition of monotone boolean functions. It should be noted however that finding the modular decomposition of a boolean function is NP-hard [12]. It was shown in [15], that computing clutters in linear time would contradict the SETH conjecture.

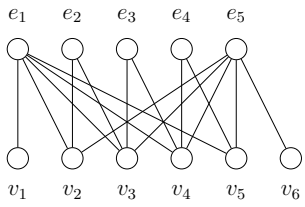


Figure 2.4: An example of a module $M = \{v_2, v_3, v_4, v_5\}$

An Application of Standard Module. If we consider that a bipartite graph is the incidence bipartite of some hypergraph then we could apply the modular decomposition of hypergraphs to decompose the bipartite graph, as can be seen in the examples of Figures 2.3, 2.4.

2.2.2 The k-subset and Courcelle's Modules

Often when generalizing graph concepts to hypergraphs there are several potential generalizations. In fact we found in the literature two variations on the hypergraph module definition: the k-subset modules defined in [13] and the Courcelle's modules defined in [26]. In this section we will first recall them and study their relationships to the standard one (Definition 2.2.1).

Definition 2.2.3. *(**k-subset module** [13]) Given a hypergraph H, we call **k-subset module** $M \subseteq V(H)$ satisfies: $\forall A, B \subseteq V(H)$ s.t. $2 \leq |A|, |B| \leq k$ and $A \cap M \neq \emptyset$, $B \cap M \neq \emptyset$ and $A \setminus M = B \setminus M \neq \emptyset$ then $A \in \mathcal{E}(H) \Leftrightarrow B \in \mathcal{E}(H)$.*

If H is a 2-uniform hypergraph (i.e., an undirected graph) the 2-subset modules are simply the usual graph modules. Families of k-subset modules also yield a partitive family. It is easy to check since the proof is fairly similar to that of standard modules. An example of a 3-subset module is given in Figure 2.5:

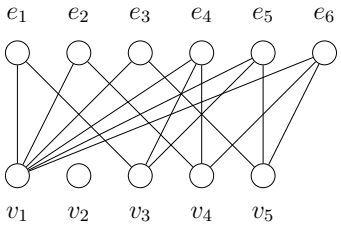
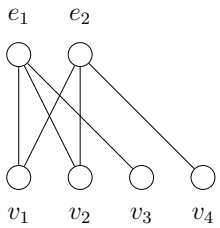


Figure 2.5: An example of a 3-subset module M=$\{v_3, v_4, v_5\}$

Proposition 2.2.3. *If M is a $|V(H)|$-subset module then M is a standard module. But the converse is false.*

Proof. $\forall A, B \in \mathcal{E}(H)$ s.t. $A \cap M \neq \emptyset$, $B \cap M \neq \emptyset$ let $A' = (A \setminus M) \cup (B \cap M)$, we have $A' \cap M \neq \emptyset$, and $A \setminus M = A' \setminus M \neq \emptyset$, so $A \in \mathcal{E}(H)$ will imply $A' = (A \setminus M) \cup (B \cap M) \in \mathcal{E}(H)$.

We now exhibit a hypergraph H that admits a standard module which is not an $|V(H)|$-subset module.



The subset $\{v_3, v_4\}$ is a standard module, but not a $|V(H)|$-subset module since $\{v_1, v_2, v_3, v_4\} \notin \mathcal{E}(H)$ □

Definition 2.2.4. *(**Courcelle's module** [26]) Given a hypergraph H, we call **Courcelle's module** $M \subseteq V(H)$ satisfies: $\forall A \in \mathcal{E}(H)$, $A \perp M$.*

We denote by $\mathcal{F}^{\perp}$ the family of subsets of V which are orthogonal to *every* element of $\mathcal{F}$. then Courcelle's modules correspond to $\mathcal{E}(H)^{\perp}$.

This notion seems to be far from the standard hypergraph module definition [74, 73], this is why we called them Courcelle's modules. Indeed, applied to graphs, the orthogonal of the edge-set is the connected components (plus the vertex-set and the singletons) of the graph, not the modules. There could be a $O(l)$ algorithm computing Courcelle's modular decomposition tree by McConnell in [69] using as a blackbox Dahlhaus algorithm [27] plus some post-treatment and has been largely simplified by [18]. Thus this is not the main focus definition in our work but we just present the related results here.

2.2.3 Basic Facts on these Module Definitions

Proposition 2.2.4. *Let H be an hypergraph and G be its incidence bipartite, if a, b are false twins in G then $\{a, b\}$ is a standard and Courcelle's module.*

Proof. Suppose there exists an edge $e \in \mathcal{E}(H)$ with $a \in H(e)$ and $b \in H(e)$. So there exists an edge ae in G and therefore since they are twins also an edge be in G, a contradiction. This says that no edge strictly overlaps $\{a, b\}$, which is thus a Courcelle's module.

□

We also notice that for k-subset modules, this property is only always true for $k = 2$.

2.3 A General Decomposition Scheme for Partitive Families

Definition 2.3.1. *For a partitive family $\mathcal{F}$ on a ground set V, using the closure by intersection of partitive families, we can define for every $A \subseteq V$, $Minmodule(A)$ as the smallest element of $\mathcal{F}$ that contains A. In particular, let us denote by $\mathcal{M}_{x,y}$ the family of all $Minmodule(\{x, y\})$ and $\forall x, y \in V$.*

Although $\mathcal{M}_{x,y}$ does not contain all $\mathcal{F}$, simply because $|\mathcal{F}|$ can be exponentiel in $|V|$ while $|\mathcal{M}_{x,y}|$ is always quadratic. In this section we propose an algorithm scheme to compute the decomposition tree of a partitive family if the only access to the family is a call of a function that computes: for every $A \subseteq V$, $Minmodule(A)$. The goal is to minimize the total number of calls. We will now show a simple way to extract the decomposition tree, i.e., the strong elements from of $\mathcal{M}_{x,y}$.

According to the definition, if a node has only two children, this node is conventionally defined to be complete. In our algorithm these nodes will be labelled as prime in this case. After constructing the tree, we can easily transform it into another convention just by labeling all nodes with only two children as complete nodes.

Theorem 2.3.1. *For every partitive family $\mathcal{F}$ over a ground set V, its decomposition tree can be computed using $O(|V|^2)$ calls to $Minmodule(\{x, y\})$, with $x, y \in V$.*

Proof. To this aim, let us choose an initial vertex x_0 and compute $Minmodule(\{x_0, x\}, \forall x \neq x_0 \in V$ and add them to a set $\mathcal{M}$. Then we add all singletons to $\mathcal{M}$. Let μ the unique path from x_0 to the root in the decomposition tree.
Claim 1: Every prime node of μ, belongs to $\mathcal{M}$.

Proof. Consider a prime strong element $A \in \mu$, it corresponds to some node of the tree which admits children $A_0, A_1, \ldots A_k$, with $k \geq 1$ in the decomposition tree. If $x_0 \in A_0$, and take $y \in A_1$, then $Minmodule(\{x_0, y\} = A$, since A is the least common ancestor in the decomposition tree. □

Claim 2: For every complete node $A \in \mu$, with children $A_0, A_1, \ldots A_k$, if $x_0 \in A_0$, then

(i) for every $1 \leq i \leq k$ the set $A_0 \cup A_i$ belongs to $\mathcal{M}$

(ii) when the elements of $\mathcal{M}$ are sorted by their size, then $A_0 \cup A_i$ appear consecutively.

Proof. (i) In fact for every $y \in A_i$, $Minmodule(\{x_0, y\}) = A_0 \cup A_i$. Note that it may be possible that $A_0 = \{x_0\}$.

(ii) If there exists a prime node P such that $\exists i, j$ such that $|A_0 \cup A_i| < |P| < |A_0 \cup A_j|$. Since $x_0 \in P$ and $x_0 \in A_0$ then P must overlap with $A_0 \cup A_i$ or $A_0 \cup A_j$, which contradicts the fact that P is a prime node that overlaps no other element in the family. □

The above arguments also show that any $Minmodule(\{x_0, y\})$ corresponds to either a prime node, or the union of two children of a complete node. So the family $\mathcal{M}$ is made up with prime nodes that overlap no other subsets and some daisies, and they all contain x_0. Daysies are these subsets $A_0 \cup A_i$, all containing A_0, the A_i's being the petals of the daisy and A_0 its center. Note that a daisy is a simple particular case of overlap component.
Now to find the decomposition tree we can apply the following algorithm:

1. Sort by size $\mathcal{M}$. In this step eliminate multiple occurencies of a subset in $\mathcal{M}$.

2. Scan this list in increasing order and checking if the new subset considered overlaps the previous, else merge it to the previous it with **complete** (label both the two as complete) and continue. After the iteration, mark every unlabelled set is with the label **prime**.

   This provides the path from x_0 to the root of the modular decomposition tree, namely: $\mu = [\{x_0\} = X_0, X_1, \ldots, X_h = V]$.

3. We repeat this procedure for vertices haven't been computed and attach its path to the tree.

Claim 3: Every node constructed is a strong module.

Proof. Assume node X, $x_0 \in X$ overlapping with some module X'. We take any element $x' \in X' \setminus X$, then $X \Delta X'$ is a module and thus $Minmodule(x_0, x') \subseteq X \Delta X'$, which overlaps with X. Thus $Minmodule(x_0, x')$ must have been merged into X, contradiction. □

The validity of the claim directly follows from Claims 1 to 3.

For Step 1 we can use any linear sorting by value algorithm, since the size of the subsets are bounded by $n = |V|$. Clearly Step 2 can be done linear time in the size of $\mathcal{M}$. So the bottleneck of complexity is the number of calls of $Minmodule(\{x, y\})$, which is bounded by n^2. □

Consequently, if computing the function $Minimal$ of a given partitive family can be done $O(p(n))$time, then the computation of the decomposition tree can be done in $O(n^2 \cdot p(n))$. Such an approach was already used for graphs in [52] to obtain the first polynomial algorithm for modular decomposition. Let us now consider how to compute this function for the three variations of hypergaph modules defined previously.

2.4 Computing Minimal-modules for Hypergraphs

For undirected graphs, computing Minimal-modules can be done via a graph search and is linear time. We will generalize this to hypergraphs for two out of the three definitions of modules, namely the standard one and the Courcelle's one.

For purpose of efficient algorithmic design, we will represent our hypergraphs using for each vertex x a list to represent $N(x)$ i.e., the edges its belongs to and symmetrically for each edge e a list to represent $H(e)$ i.e., the vertices it contains. For a hypergraph this yields a representation using $O(n + m + l)$ memory. If the hypergraph is simple one can notice that $O(n + m + l) = O(l)$.

2.4.1 Standard Modules

Definition 2.4.1. *For a set $C \subsetneq V(H)$, an edge $A \in \mathcal{E}(H)$ is a **edge-splitter** for C, if $A \setminus C \neq \emptyset$ and $A \cap C \neq \emptyset$ and if there exists $B \in \mathcal{E}(H)$ s.t. $B \cap C \neq \emptyset$, and $(A \setminus C) \cup (B \cap C) \notin \mathcal{E}(H)$.*

In other words, a set of vertices is a standard module iff it admits no edge-splitter.

Let us now consider how to incrementally compute $Minmodule(W)$ for every subset W.

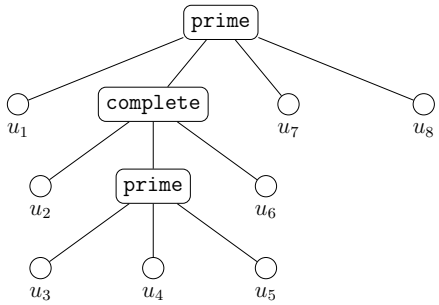
Proposition 2.4.1. *If $A \in \mathcal{E}(H)$ is a edge-splitter of $C \subseteq V(H)$ respect to B as Definition 2.4.1. Let $B' \in \mathcal{E}(H)$ be the edge such that $B' \cap C = B \cap C$ and with $|(B' \setminus C)\Delta(A \setminus C)|$ minimum. Let $X' = (B' \setminus C)\Delta(A \setminus C)$, then there is no module Y of H such that $C \subsetneq Y$ but $X' \nsubseteq Y$.*

Proof. Assume there exists a module Y such that $C \subsetneq Y$ but $X' \nsubseteq Y$.

With the choice of X', $A \cap Y \neq \emptyset$, therefore $A' = (A \setminus Y) \cup (B' \cap Y) \in \mathcal{E}(H)$.

$(B' \setminus C)\Delta(A \setminus C) = ((B'\Delta A) \cap (Y \setminus C)) \cup (B' \setminus Y)\Delta(A \setminus Y))$

u_1	1	1	1	1	1	1	0	1	1
u_2	1	1	1	1	1	1	0	0	0
u_3	0	1	1	0	1	1	0	0	0
u_4	1	0	0	1	0	0	0	0	0
u_5	1	0	1	1	0	1	0	0	0
u_6	0	0	0	1	1	1	0	0	0
u_7	0	0	0	0	0	0	1	1	1
u_8	1	1	1	1	1	1	0	0	1



x	minimal module of x and others
u_1	$\{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8\}$
u_2	$\{u_2, u_6\}, \{u_2, u_3, u_4, u_5\}, \{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8\}$
u_3	$\{u_3, u_4, u_5\}, \{u_2, u_3, u_4, u_5\}, \{u_3, u_4, u_5, u_6\}, \{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8\}$
u_4	$\{u_3, u_4, u_5\}, \{u_2, u_3, u_4, u_5\}, \{u_3, u_4, u_5, u_6\}, \{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8\}$
u_5	$\{u_3, u_4, u_5\}, \{u_2, u_3, u_4, u_5\}, \{u_3, u_4, u_5, u_6\}, \{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8\}$
u_6	$\{u_2, u_6\}, \{u_3, u_4, u_5, u_6\}, \{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8\}$
u_7	$\{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8\}$
u_8	$\{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8\}$

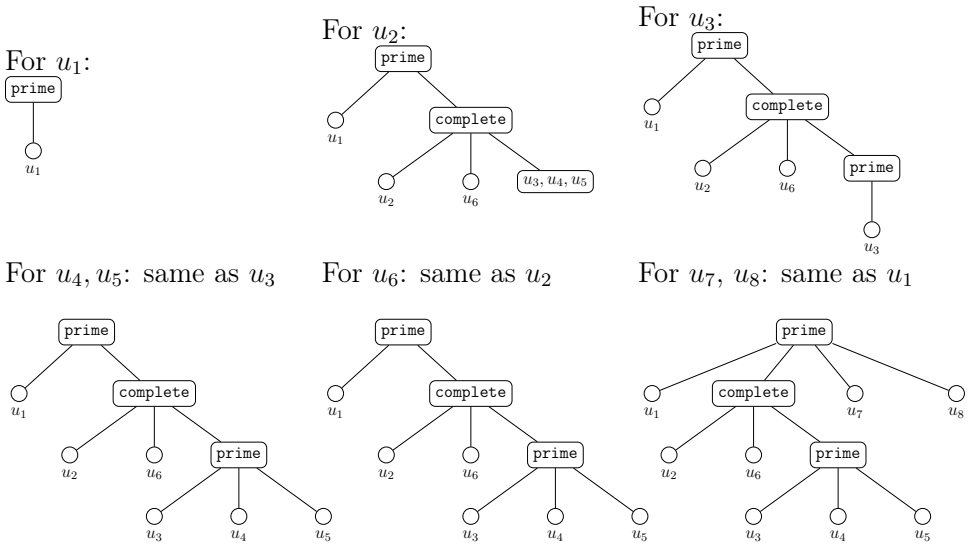


Figure 2.6: Computing modular decomposition tree based on minimal modules

$(A' \setminus C)\Delta(A \setminus C) = (B'\Delta A) \cap (Y \setminus C) \subseteq (B' \setminus C) \setminus (A \setminus C)$

If $(B' \setminus Y)\Delta(A \setminus Y) = \emptyset$ then $X' = (B' \setminus C)\Delta(A \setminus C) \subseteq Y$ a contradiction.

Else $|(A' \setminus C)\Delta(A \setminus C)| < |(B' \setminus C) \setminus (A \setminus C)|$. Since $A' \in \mathcal{E}(H)$ and $C \subsetneq Y$, then $A' \cap C = B' \cap C$ this yields a contradiction with the definition of B', we should have taken A' instead of B'.

So in both cases a contradiction was found, therefore such a module Y cannot exist. □

Let us now turn these ideas into an algorithm, using lexicographic orderings of the edges, % means comments in the following pseudo-code.

Algorithm 1 Modular-closure

```
1: procedure MODULAR-CLOSURE(H a simple hypergraph and W ⊊ V(H))
2:     τ ← a lexicographic ordering of E(H) w.r.t an arbitrary ordering of V
3:     C ← W, X ← W
4:     Compute the induced hypergraph H(C)
5:     for 1 ≤ i ≤ |E(H(C))| do
6:         Compute the ordered lists L_i of the restriction to V(G) \ C of the edges
   in E(H) that contain f_i ∈ E(H(C))
7:     end for
8:     Q(C) ← {L_1, ..., L_{|E(H(C))|}} the ordered partition made up with these lists
9:     if |Q(C)| = 1 then
10:        C is a module, STOP
11:    else
12:        L ← First(Q(C))        // the first class in the ordered partition
13:    end if
14:    while Next(L) ≠ NIL do        // the next element of L in the ordered
   partition
15:        X ← Comparison(L, Next(L))
16:        if X = ∅ then
17:            L ← Next(L)
18:        else      //if L has been split during the update we take its first part
19:            C ← C ∪ X, update Q(C) via partition refinement with X,
20:            L ← First(L)
21:        end if
22:    end while
23:    RESULT ← C        //C is a minimal module that contains W
24: end procedure
```

Theorem 2.4.1. *If H is a simple hypergraph, for every set $W \subseteq V(H)$, Algorithm Modular-closure computes $Minmodule(W)$ in $O(n \cdot l)$. And its modular decomposition tree can be computed in $O(n^3 \cdot l)$.*

Proof. (i) **Correctness:** First we notice that C is a module of H iff all the lexicographically sorted lists L_i are equal. At each step of the lexicographic process a list can only be cut into parts, no lists are merged. If at some step of the algorithm two

Algorithm 2 Procedure Comparison

1: **procedure** COMPARISON(2 lists L', L'' of the restriction to $V(G) \setminus C$ of the edges in $\mathcal{E}(H)$ that contain some $f \in \mathcal{E}(H(C))$. They are supposed to be lexicographically increasingly ordered using τ)
2: **if** $L' = L''$ **then** $X \leftarrow \emptyset$, STOP
3: **else**
4: Let $e \in L'$ and $f \in L''$ be the first lexicographically difference
5: **end if**
6: **if** $(e <_\tau f) or (e \neq \emptyset$ and $f = \emptyset)$ **then** // $e \notin L''$ is a edge-splitter
7: compute $f' \in L''$ that minimizes $|h(e)\Delta h(f')|$ with $f' \in L''$
8: $X \leftarrow h(e)\Delta h(f')$
9: **else** // $(e >_\tau f) or (e = \emptyset$ and $f \neq \emptyset)$, i.e. $f \notin L'$ is a edge-splitter
10: compute $e' \in L'$ that minimizes $|h(f)\Delta h(e')|$ with $e' \in L'$
11: $X \leftarrow h(f)\Delta h(e')$
12: **end if** // Note that $e = \emptyset$, $f \neq \emptyset$ (resp. $e \neq \emptyset$, $f = \emptyset$) corresponds to the case $|L| < |Next(L)|$ (resp. $|L| > |Next(L)|$)
13: **end procedure**

lists L_i, L_j are equal, and if afterwards they are cut into sublists via the refinement process, equality between sublists is preserved since the refinement act similarly on the lists. Thus the algorithm scan the lists form left to right using a single sweep and the following invariant: at each step of the while loop all the lists before the current list L are all equal to L.

Using the procedure Comparison either the lists are equal and then we proceed else using Property 2.4.2 we know that we can add this set of vertices. At the end of the algorithm either all lists are equals and $C \neq V(H)$ and therefore C is the non trivial minimal module containing W or $C = V(H)$ and there is no other module between W and $V(H)$.

(ii) **Complexity Analysis:** To implement the first step (line 1) we can use an ordered partition refinement technique on $\mathcal{E}(H)$ (see [46]) using the sets $N(x)$ for every $x \in V(H)$ as pivot sets. This provides a total ordering τ of $\mathcal{E}(H)$. This can be done in $O(n + m + l)$.

To compute $\mathcal{Q}(C)$, we can use the same ordered partition refinement technique using the sets $N(x)$ for every $x \in C$ as pivot sets we can compute the ordered partition of $\mathcal{E}(H)$. Starting from the partition $P_0 = \{\mathcal{E}(H)\}$, we refine this partition successively using $N(x_i)$ for every $x_i \in C$. Let us denote by P_f the partition obtained after this round of refinements. Each part of P_f can be ordered using τ, since partition refinement can maintain an initial ordering of its elements within the same complexity. So if we start with the initial ordering τ in the unique part of P_0. And the parts are lexicographically ordered with respect to their intersection to C. This can be done in $O(|C| + \Sigma_{x \in C}|N(x)|)$.

In fact after line 6 we can ignore the vertices of C, a similar remark holds when C is updated.

Now we have to check if all edges lists L_i are identical or not and stop at the first difference. Since the lists are ordered lexicographically using an ordering τ of

the vertices, a simple scan of these ordered lists is enough to compute (Comparison procedure) of Algorithm 1.

When C and $\mathcal{Q}(C)$ are updated, the algorithm goes on with the first part of the previous current list L. First means that if L has been split during the update we take its first part. Therefore in the worst case some list can be analyzed several times (at most n times) and therefore the overall complexity of the list scan is bounded by $O(n \cdot l)$.

When a difference is found between two lists we have to search for an edge that minimizes the symmetric distance with respect to the differentiating edge. Even though it can be done several times for a given edge, but every time we launch this search, at least one vertex will be added into C, thus at most search for n times. So the overall complexity of these searches is $O(n \cdot l)$.

Therefore the whole process is in $O(n + m + l + n \cdot l) = O(n \cdot l)$. Using Theorem 2.3.1 we obtain the decomposition tree in $O(n^3 \cdot l)$. □ □

Up to our knowledge, [74] states there is a polynomial time decomposition algorithm for clutters based on its $O(n^4m^3)$ modular closure algorithm without precising the complexity, our algorithm is an improvement because our total decomposition time is already smaller than $O(n^4m^3)$.

Corollary 2.4.1. *For every simple hypergraph H, its modular decomposition tree can be computed in $O(n^3 \cdot l)$.*

Proof. We can use Algorithm 1 to compute in $O(l)$ for a pair of vertices x, y $Minmodule(\{x, y\})$. Therefore the process described in Theorem 2.3.1 provides a decomposition tree using n^2 calls to this procedure. This leads to an algorithm in $O(n^3 \cdot l)$. To transform it into a hypergraph modular decomposition tree, it only remains to precise the labels "complete" into series ou parallel, which can be done easily by a test on the edges. □

Remark: It could be possible that for simple hypergraphs, linear time modular decomposition exists, i.e. in $O(l)$ time, as is the case for graphs (for a survey on graph algorithms see [47]).

2.4.2 Algorithms for the Courcelle's Modules

Although we already have a linear time algorithm using orthogonality, it is worth showing that this decomposition also fits in our general framework for partitive families. Reformulating the Courcelle's module definition we have:

Lemma 2.4.1. *Given a set $M \subseteq V(H)$, a set $X \subseteq V(H) \setminus M$ is a Courcelle-splitter of M if there exists an edge $e \in \mathcal{E}(H)$ such that: e overlaps M and $X = H(e) \setminus M$. A subset M is a Courcelle's module if there is no Courcelle-splitter for M.*

Using this lemma, we can apply to this decomposition the same framework we developed for standard modules.

Theorem 2.4.2. *The minimal Courcelle's module $Minmodule(A)$ that contains a given set $A \subseteq V(H)$ can be computed in $O(n + m + l)$ time.*

Proof. The correctness of the algorithm comes from Lemma 2.4.1 We can still use the structure of Algorithm 1, with a little change in the test of line 9-22 which must be modified in the following way.

---

while there is an edge e that overlap with W **do**
  $X \leftarrow H(e) \setminus W$, $C \leftarrow C \cup X$
end while

This modification is easier to implement, and thus the complexity analysis of complexity we gave in the previous section also holds here. □

2.4.3 Decomposition into k-subset Modules

Let us now consider the k-subset modules as defined in [13]. For this specific definition of hypergraph modules, we need a to implement an efficient **dynamic partition refinement** tool.

Definition 2.4.2. *A subset $X \neq \emptyset$ is a **k-splitter** of the set M if there exist $A, B \subseteq V$ s.t. $2 \leq |A|, |B| \leq k$ and $A \cap M \neq \emptyset$, $B \cap M \neq \emptyset$ and $A \setminus M = B \setminus M = X$, $A \in \mathcal{E}(H)$ but $B \notin \mathcal{E}(H)$.*

Proposition 2.4.2. *If $X \subseteq V(H)$ is a k-splitter of $C \subseteq V(H)$ respect to A, B as above. then there is no module Y of H such that $C \subsetneq Y$ but $X \nsubseteq Y$.*

Proof. Assume there exists a module Y such that $C \subsetneq Y$ but $X' \nsubseteq Y$.

We consider the edges A, B above.
Since $C \subsetneq Y$, then $A \cap Y \neq \emptyset$, $B \cap Y \neq \emptyset$
Since $X \nsubseteq Y$, then $A \setminus Y = B \setminus Y = X \setminus Y \neq \emptyset$
Thus since Y a module $A \in \mathcal{E}(H)$ implies $B \in \mathcal{E}(H)$, a contradiction. Therefore such a module Y cannot exist. □

Lemma 2.4.2. *Given a set $M \subseteq V(H)$, all k-subset splitters of M, if exist, are in the form of $H(e) \setminus M$ for some $e \in \mathcal{E}(H)$, $|H(e)| \leq k$.*

Proof. If X is a k-subset splitter of M, take the edge $A \in \mathcal{E}(H)$ in the definition of k-splitter, then clearly $X = h(A) \setminus M$. □

Such an edge will be called an **edge-splitter** of M.

Let us define by $D(k, h) = \Sigma_{i=1}^{i=k} \binom{h}{i}$ for $1 \leq k \leq h$, where $\binom{h}{i}$ denotes the usual binomial coefficient, i.e. the number of subsets of size i in a set of size h. By convention all values of $D(k, h)$ strictly greater that $|\mathcal{E}(H)|$ will be fixed as **Out-of-Range**, considered as a huge number.

Lemma 2.4.3. *For a simple hypergraph H, given a set $M \subseteq V(H)$ and an edge $e \in \mathcal{E}(H)$ s.t. $|H(e)| \leq k$, $e \cap M \neq \emptyset$ and $X = e \setminus M \neq \emptyset$, let L be the list of edges in $\mathcal{E}(H)$ with size $\leq k$ and whose intersection with $V(H) \setminus M$ are identical to X and intersection with M is not empty, i.e. $L = \{e' \in \mathcal{E}(H) \mid e' \setminus M = X,\ e' \cap M \neq \emptyset$ and $|H(e')| \leq k\}$. If $|L| < D(k - |X|, |M|)$ then e is an edge-splitter of M.*

Proof. It's equivalent to check for such an e given above and $X = e \setminus M$, whether every non empty subset B of size $\leq k - |X|$ in M has $X \cup B \in \mathcal{E}(H)$. Since H is simple and there are no identical elements in L, a counting argument captures the condition. Moreover, in this way the number of subsets checked is $\leq |\mathcal{E}(H)|$. □

Lemma 2.4.4. *Given a pair of vertices x, y, we can compute in $O(|\mathcal{E}(H)|^2)$ time the minimal non trivial k-subset module that contains x and y.*

Algorithm 3 k-subset modular-closure

```
1: procedure K-SUBSET MODULAR-CLOSURE(H a simple hypergraph, k an integer
   such that 1 ≤ k ≤ |V(H)| and W ⊊ V(H))
2:     Compute all D(k, h) for |W| ≤ h ≤ |V(H)|
3:     C ← W, X ← W
4:     while X ≠ ∅ do
5:         Compute the induced hypergraph H(V(G) \ C)
6:         Compute a lexicographic ordering τ of E(H) with respect to some ordering
   of the vertices, s.t. priority of vertices in V(G) \ C are higher than in C
7:         E' ← {e_i ∈ E(H) overlap C, |H(e_i)| ≤ k}
8:         for 1 ≤ i ≤ |E'(H(V(G) \ C))| do
9:             Compute the ordered lists L_i of the restriction to V(G) \ C of the
   edges in E'(H) that contain f_i ∈ E'(H(V(G) \ C))
10:        end for
11:        if For some i, |L_i| < D(k − |f_i|, |C|) then
12:            X ← X ∪ (H(e_i) \ C)
13:            if V(G) = C∪X then        // there is no non-trivial module between
   W and V(H)
14:                RESULT ← V(H), STOP
15:            else      // X is a splitter for C
16:                C ← C ∪ X
17:            end if
18:        else
19:            X ← ∅
20:        end if
21:        RESULT ← C       // C is a non trivial module containing W
22:    end while
23: end procedure
```

Theorem 2.4.3. *For a simple hypergraph H and $A \subsetneq V(H)$, for any fixed integer $k \leq |V(H)|$, algorithm 3 (k-subset modular-closure) can compute the minimal k-subset module that contains A in $O(n \cdot l)$ time.*

Proof. The correctness of the previous algorithm directly follows from Lemma 2.4.3. The computation of the $D(k, j)'s$ can be done in $O(n^2)$ using Pascal's triangle and does not require big numbers encoding using the symbol Out-of-Range.

The bottleneck of the complexity is in line 7-8, with the computation of the lists L_i. It can be done in $O(l)$.

So the overall complexity is in $O(n^2 + n \cdot m + n \cdot l) = O(n \cdot l)$. □

Corollary 2.4.2. *For a simple hypergraph H and for any integer $k \leq |V(H)|$, its k-subset modular decomposition tree can be computed in $O(n^3 \cdot l)$.*

Proof. We can use the complete framework introduced in this paper. Using the algorithm proposed in Theorem 2.3.1 to compute the k-subset modular decomposition tree. So we need to call n^2 times the previous algorithm 2, which yields the announced complexity.

It should be noticed that the computation of the $D(k,j)'s$ can be done as a preprocessing and costs $O(n^2)$ only once.

□

2.5 Two Negative Results: ϵ-module and ϵ-splitter module

When working with modules, especially on graphs that rise from large networks, it is natural to look for subsets of vertices that behave almost like a module. Suppose we relax the constraint of *exactly* the same neighborhood in order to have a tolerance of errors. Thus, instead of having X (resp. Y) as a complete bipartite to a module M (resp. no edges to M), we allow some bounded number of non-edges (resp. of edges). This relaxed notion of modular decomposition has indeed been used in biology or social science for weighted bipartite graphs [35, 34]. However, few theoretical results have been showed, and results on a larger scope of graph classes also remain to be explored.

Here we try to accept some errors of at most k edges, for some fixed integer k, could be missing in the complete bipartite between M and $N(M)$, and symmetrically that at most k edges can exist between M and $\overline{N(M)}$. But doing so we loose most of the nice algebraic properties of modules – in particular that overlapping modules are closed under intersection, union and difference [19]. Furthermore most algorithms for modular decomposition are based on these algebraic properties.

Another natural idea is to relax the condition on the complete bipartite between M and $N(M)$, for example asking for a graph that does not contain any $2K_2$. Unfortunately as shown in [80] to test wether a given graph admits such a decomposition is NP-complete. In fact they studied a generalized join decomposition solving a question asked in [49] studying perfection. This is why the following generalization of module defined for any integer ϵ[2], seems to be a good compromise:

Definition 2.5.1. *A subset $M \subseteq V(G)$ is an ϵ-module if $\forall x \in V(G) \setminus M$, either $|M \cap N(x)| \leq \epsilon$ or $|M \cap N(x)| \geq |M| - \epsilon$*

In other words, we tolerate ϵ edges of errors per node outside the ϵ-module, and not ϵ errors per module. It should be noticed that with $\epsilon = 0$, we recover the usual definition of modules [47], i.e., $\forall x \in V(G) \setminus M$, either $M \cap N(x) = \emptyset$ or $M \cap N(x) = M$. Necessarily we will only consider $\epsilon < |V(G)| - 1$. We can easily verify that $V(G)$ and $\forall A \subseteq V(G)$ such that $|A| \leq 2 \cdot \epsilon + 1$ are always ϵ-modules, they are called **trivial** ϵ-modules.

[2]We use ϵ to denote small error, despite being greater than 1.

Then we check if ϵ-modules are partitive:

Proposition 2.5.1. *Let $A, B \subseteq V(G)$ be two overlaping non trivial ϵ-modules, then there could be $c = \Omega(\min(|A|, |B|))$, s.t. $A \cup B$ is not an ϵ-module, for all $\epsilon \leq c$.*

Proposition 2.5.2. *Let $A, B \subseteq V(G)$ be two overlaping non trivial ϵ-modules, then there could be $c = \Omega(n)$, s.t. A-B is not an ϵ-module, for all $\epsilon \leq c$.*

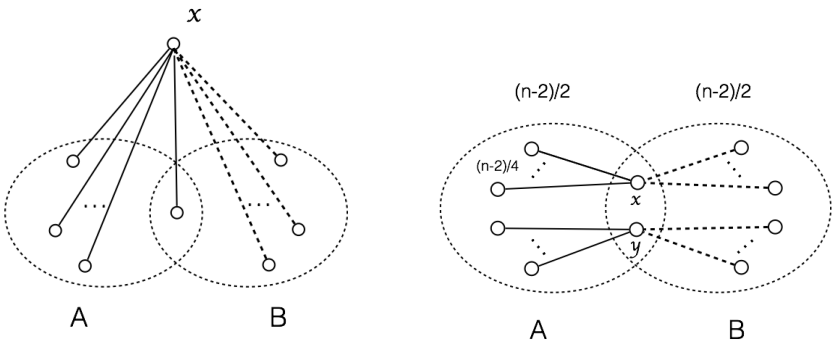


Figure 2.7: Example for Proposition 2.5.1 and 2.5.2

For standard modular decomposition the notion of strong modules as modules that does not overlap with any other is central. For ϵ-modular decomposition we can observe that there is no strong modules other than V and $\{v\}, v \in V$ that are strong $\epsilon-$modules. The reason is that, for $\epsilon \geq 1$, any subset of vertices of size 2 is a trivial ϵ-module, then assume there is a classical strong module $V_1 \neq V$, $|V_1| > 1$, then take any vertex $v \in V_1$ and any vertex $u \in V \setminus V_1$, then $\{u, v\}$ is a ϵ-module and overlapping with V_1.

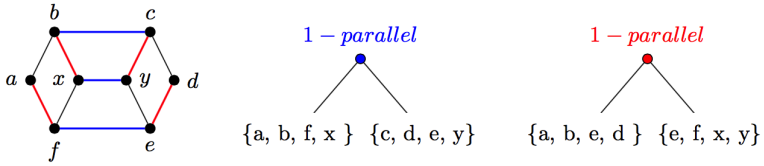


Figure 2.8: Two ways of 1-parallel decomposition of a graph

The decomposition tree is then not well-defined for ϵ-module, the ways to decompose a graph could be non-unique, and we here show that even a weaker purpose of decomposition for one step is still hard.

Definition 2.5.2. *For a graph G with $|V(G)| \geq 2\epsilon + 3$, we say that G admits an ϵ-series (resp. ϵ-parallel) decomposition if there exists a partition of $V(G)$, $\mathcal{P} = \{V_1, \ldots V_k\}$ such that: $\forall i$, $1 \leq i \leq k$, $|V_i| \geq 2\epsilon + 1$ and $\forall x \in V_i$ and for every $j \neq i$, x is adjacent (resp. non-adjacent) to all vertices of V_j with perhaps ϵ errors.*

For $\epsilon = 1$ the problem of recognizing if a graph admits an 1-parallel decomposition could be reduced to the matching cut-set problem that is known to be NP-hard [21, 71, 14] and therefore we have:

Theorem 2.5.1. *Finding if a graph admits an 1-parallel decomposition is NP-hard.*

Proof. Let G be a graph with minimum degree 3, and suppose that it admits an 1-parallel decomposition into $V_1, \ldots, V_k$. Necessarily $\forall i$, $|V_i| > 1$, since there is no pending vertex. Therefore $\{V_1, \cup_{1<i\leq k} V_i\}$ is a matching cut set of G. So using [21], deciding if a graph admits 1-parallel decomposition is NP-complete. □

Let us now study an alternative way of defining approximate modules. For any integer ϵ one can define:

Definition 2.5.3. *A subset $M \subseteq V(G)$ is an ϵ-splitter module if there are at most ϵ splitters in $V(G) \setminus M$.*

It should be noticed that with $\epsilon = 0$ this definition gives back the usual module definition [47], i.e., $\forall x \in V(G) \setminus M$, either $M \cap N(x) = \emptyset$ or $M \cap N(x) = M$. $\forall A \subseteq V(G)$ such that $|A| \leq 1$ or $|A| \geq |V| - \epsilon$ then A is an ϵ-splitter module. We call such a set A a trivial ϵ-splitter module.

The family of ϵ-splitter modules of a graph satisfies:

Proposition 2.5.3. *Let $A, B \subseteq V(G)$ be two overlaping non trivial ϵ-splitter modules s.t. $A \cap B \neq \emptyset$ then: $A \cap B$ could be an 2ϵ-splitter module.*

For this definition, it is not closed under intersection. Therefore we cannot easily define a closure operator with this notion. Thus it is even harder (nearly impossible) to define a decomposition notion on it.

2.6 Conclusions of Chapter

This chapter present our study the generalization of modular decompositions and their time-efficient algorithms. First we recalled the basic concept of partitive families and the relationship with unique decomposition.

We then presented our work on modular decomposition in hypergraphs for three definitions from the literature: the standard modules [73], the k-subset modules [13] and the Courcelle's modules [26]. We obtained positive results for the three as they all lead to partitive families. We developed a general algorithmic scheme to compute their modular decomposition following the idea in [52] for modules in graphs, by implementing the appropriately functions in this scheme, we got a $O(n^3 \cdot l)$ algorithm for modular decomposition of standard module and k-subset module, improving the previous result based on a $O(n^4 m^3)$ algorithm [74] for standard module and $O(n^{3k-5})$ algorithm [13] for k-subset module respectively, and conclude the decomposition of k-subset module in hypargraphs is in P.

We also presented negative results for two generalization by allowing errors in graph modules: ϵ-module and ϵ-splitter module, we conclude that they are not partitive thus do not satisfy the condition of the unique modular decomposition theorem. More over, testing of one-step parallel decomposition ϵ-module for $\epsilon = 1$ is already NP-hard. This shows that allowing errors in graph modules will destruct the validity of a unique tree representation.

Results of this chapter have been presented in:

- Michel Habib, Fabien de Montgolfier, Lalla Mouatadid, Mengchuan Zou:
  A General Algorithmic Scheme for Modular Decompositions of Hypergraphs and Applications. IWOCA 2019, invited to a special issue of Journal *Theory of Computing Systems*

- Michel Habib, Lalla Mouatadid, Mengchuan Zou:
  Approximating Modular Decomposition is Hard.
  Accepted to The International Conference on Algorithms and Discrete Applied Mathematics (CALDAM) 2020.

The second publication here presents also some positive results of enumerating all minimal ε-modules, which are not included in this thesis.

Perspective We are interested in the following future topics related to our work:

- Can we define graph classes based on these definition of modules in hypergraphs? What properties will they have?

- For graphs, is there other generalizations of modules that have good decomposition properties? For ϵ-module for $\epsilon = 1$ we could define 1-cograph, how to characterize its properties?

Chapter 3

Strategy for Generalized Binary Search in Weighted Trees

3.1 Introduction

3.1.1 The Problem

We present in this chapter the generalized binary search problem in weighted trees, that is to locate a "hidden" target node in a weighted tree by asking queries with costs to the generalized binary search oracle. The generalized binary search model follows the idea of binary search, which is indeed equivalent to search for a target node in a path, each query selects a node, and the oracle replies which 'side' (or sub-path) of the queried node the target node belongs to. The model generalize the binary search both for its searching space as a graph and also the non-uniform time cost for queries. The problem can be stated as follows. Given a node-weighted input tree T (in which the query time of a node is provided as its weight), design a search strategy (sometimes called a decision tree) that locates a hidden *target node* x by asking *queries*. Each query selects a node v in T and after the time that equals the weight of the selected node, a reply is given: the reply is either 'yes' which implies that v is the target node and thus the search terminates, or it is 'no' in which case the search strategy receives the edge outgoing from v that belongs to the shortest path between u and v. The goal is to design a search strategy that locates the target node and minimizes the search time in the worst case.

3.1.2 Related Works

In this work we focus on the worst case search time for a given input graph and we only remark that other optimization criteria has been also considered [22, 57, 56, 86]. For other closely related models and corresponding results see e.g. [3, 48, 59, 64, 85].

The node-query model. The vertex search problem is more general than its 'edge variant' that has been more extensively studied. In the latter problem one selects an edge e of an edge-weighted tree $T = (V, E, w)$ in a query and learns in which of the two components of $T - e$ the target node is located. Indeed, this edge

variant can be reduced to our problem as follows: first assign a 'large' weight to each node of T (for example, one plus the sum of the weights of all edges in the graph) and then subdivide each edge e of T giving to the new node the weight of the original edge, $w(e)$. It is apparent that an optimal search strategy for the new node-weighted tree should never query the nodes with large weights, thus immediately providing a search strategy for the edge variant of T.

An optimal search strategy can be computed in linear-time for an unweighted tree [76, 81]. The number of queries performed in the worst case may vary from being constant (for a star one query is enough) to being at most $\log_2 n$ for any tree [76] (by always querying a node that halves the search space). Several following results have been obtained in [37]. First, it turns out that $\log_2 n$ queries are always sufficient for general simple graphs and this implies a $O(m^{\log_2 n} n^2 \log n)$-time optimal algorithm for arbitrary graphs. The algorithm which performs $\log_2 n$ queries also serves as a $O(\log n)$-approximation algorithm, also for the weighted version of the problem. On the other hand, it has been proven that optimal algorithm with a running time of $O(n^{o(\log n)})$ is in contradiction with the Exponential-Time-Hypothesis, and for $\varepsilon > 0$, $O(m^{(1-\varepsilon)\log n})$ is in contradiction with the Strong Exponential-Time-Hypothesis. When non-uniform query times are considered, the problem becomes PSPACE-complete. Also, a generalization to directed graphs also turns out to be PSPACE-complete.

We also refer the interested reader to further works that consider a probabilistic version of the problem, where the answer to a query is correct with some probability $p > \frac{1}{2}$ [8, 37, 38, 53]. In particular, for any $p > \frac{1}{2}$ and any undirected unweighted graph, a search strategy can be computed that finds the target node with probability $1-\delta$ using $(1-\delta)\frac{\log_2 n}{1-H(p)} + o(\log n) + O(\log^2 \frac{1}{\delta})$ queries in expectation, where $H(p) = -p\log_2 p - (1-p)\log_2(1-p)$ is the entropy function. See [79] for a model in which a fixed number of queries can be answered incorrectly during a binary search.

The edge-query model. In the case of unweighted trees, an optimal search strategy can be computed in linear time [60, 72]. (See [29] for a correspondence between edge rankings and the searching problem.) The problem of computational complexity for weighted trees attracted a lot of attention. On the negative side, it has been proved that it is strongly NP-hard to compute an optimal search strategy [28] for bounded diameter trees, which has been improved by showing hardness for several specific topologies: trees of diameter at most 6, trees of degree at most 3 [23] and spiders [24] (trees having at most one node of degree greater than two). On the other hand, polynomial-time algorithms exist for weighted trees of diameter at most 5 and weighted paths [23]. We note that for weighted paths there exists a linear-time but approximate solution given in [56]. For approximate polynomial-time solutions, a simple $O(\log n)$-approximation has been given in [28] and a $O(\log n/\log\log\log n)$-approximate solution is given in [23]. Then, the best known approximation ratio has been further improved to $O(\log n/\log\log n)$ in [24].

Some bounds on the number of queries for unweighted trees have been developed. Observe that an optimal search strategy needs to perform at least $\log_2 n$ queries in the worst case. However, there exist trees of maximum degree Δ that require $\Delta \log_{\Delta+1} n$ queries [7]. On the other hand, $\Theta(\Delta \log n)$ queries are always

sufficient for each tree [7], which has been improved to $(\Delta+1)\log_\Delta n$ [58], $\Delta\log_\Delta n$ [30] and $1+\frac{\Delta-1}{\log_2(\Delta+1)-1}\log_2 n$ [37].

We also point out that the considered problem, as well as the edge variant, being quite fundamental, were historically introduced several times under different names: minimum height elimination trees [78], ordered colourings [54], node and edge rankings [50], tree-depth [75] or LIFO-search [43].

Table 3.1 summarizes the complexity status of both node- and edge-query models (in case of unweighted paths in both cases the solution is the classical binary search algorithm) and places our result in the general context.

Table 3.1: Computational complexity of the search problem in different graph classes, including our results for weighted trees. Completeness results refer to the decision version of the problem.

Graph class	*Unweighted*	*Weighted*
Paths:	exact in $O(n)$ time	exact in $O(n^2)$ time [23]
Trees:	exact in $O(n)$ time [76, 81]	strongly NP-complete [31] $O(\sqrt{\log n})$-approx. in poly-time (Thm. 3.4.1) $(1+\varepsilon)$-approx. in $n^{O(\log n/\varepsilon)}$ time (Thm. 3.5.1)
Undirected:	exact in $n^{O(\log n)}$ time [37] $O(\log n)$-approx. in poly-time [37]	PSPACE-complete [37] $O(\log n)$-approx. in poly-time [37]
Directed:	PSPACE-complete [37]	PSPACE-complete [37]

Searching partial orders. The problem of searching a partial order with uniform query times is NP-complete even for partial orders with maximum element and bounded height Hasse diagram [17, 29]. For some algorithmic solutions for random partial orders see [17]. For a given partial order P with maximum element, an optimal solution can be obtained by computing a branching B (a directed spanning tree with one target) of the directed graph representing P and then finding a search strategy for the branching, as any search strategy for B also provides a feasible search for P [29]. Since computing an optimal search strategy for B can be done efficiently (through the equivalence to the edge-query model), finding the right branching is a challenge. This approach has been used in [29] to obtain an $O(\log n/\log\log n)$-approximation polynomial time algorithm for partial orders with maximum element.

We remark that searching a partial order with maximum element or with minimum element are essentially quite different. For the latter case a linear-time algorithm with additive error of 1 has been given in [76]. As observed in [29], the problem of searching in tree-like partial orders with maximum element (which corresponds to the edge-query model in trees) is equivalent to the edge ranking problem.

3.1.3 Organization of the Chapter

Section 3.2 explains the necessary notation and a formal statement of the problem. For our analysis, Section 3.3 modelizes our problem into a model more similar to combinatorial optimization problem than the natural description of strategies. Precisely, Section 3.3.1 restates the problem in such a way that with each vertex v of the input tree we associate a sequence of vertices that need to be iteratively queried when v is the root of the current subtree that contains the target node. In Section 3.3.2 we extend this approach by associating with each vertex a sequence of not only vertices to be queried but also time points of the queries.

The latter problem formulation is suitable for a dynamic programming algorithm provided in Section 3.4. In this section we introduce an auxiliary, slightly modified measure of the cost of a search strategy. First we provide a quasi-polynomial time dynamic programming scheme that provides an arbitrarily good approximation of the output search strategy with respect to this modified cost (the analysis is deferred to Section 3.4.3), and then we prove that the new measure is sufficiently close to the original one (the analysis is deferred to Section 3.4.5). These two facts provide the quasi-polynomial time scheme for the tree search problem, achieving a $(1+\varepsilon)$-approximation with a computation time of $n^{O(\log n/\varepsilon^2)}$, for any $0 < \varepsilon < 1$.

In Section 3.5 we observe how to use the above algorithm to derive a polynomial-time $O(\sqrt{\log n})$-approximation algorithm for the tree search problem. This is done by a divide and conquer approach: a sufficiently small subtree T^* of the input tree T is first computed so that the quasi-polynomial time algorithm runs in polynomial (in the size of T) time for T^*. This decomposes the problem: having a search strategy for T^*, the search strategies for $T - T^*$ are computed recursively. Details of the approach are provided in Section 3.5.2.

3.2 Preliminaries

3.2.1 Notation and Query Model

We now give a more formal description of the problem of searching of an unknown target node x by performing queries on the vertices of a given node-weighted rooted tree $T = (V, E, w)$ with weight function $w\colon V \to \mathbb{R}_+$. Each *query* selects one vertex v of T and after $w(v)$ time units receives an answer: either the query returns *true*, meaning that $x = v$, or it returns a neighbor u of v which lies closer to the target x than v. Since we assume that the queried graph T is a tree, such a neighbor u is unique and is equivalently described as the unique neighbor of v belonging to the same connected component of $T \setminus \{v\}$ as x.

All trees we consider are rooted. Given a tree T, the root is denoted by $r(T)$. For a node $v \in V$, we denote by T_v the subtree of T rooted at v. For any subset $V' \subseteq V$ (respectively, $E' \subseteq E$) we denote by $T[V']$ (resp., $T[E']$) the minimal subtree of T containing all nodes from V' (resp., all edges from E'). For $v \in V$, $N(v)$ is the set of neighbors of v in T.

For $U \subseteq V$ and a target node $x \notin U$, there exists a unique maximal subtree of $T \setminus U$ that contains x; we will denote this subtree by $T\langle U, x\rangle$.

We denote $|V| = n$. We will assume w.l.o.g. that the maximum weight of a vertex is normalized to 1. (This normalization is immediately obtained by a proportional scaling of all units of cost.) We will also assume w.l.o.g. that the weight function satisfies the following *star condition*:

$$\text{for all } v \in V,\ w(v) \leq \sum_{u \in N(v)} w(u).$$

Observe that if this condition is not fulfilled, i.e., for some vertex v will have $w(v) > \sum_{u \in N(v)} w(u)$, then vertex v will never be queried by any optimal strategy in v, since a query to v can then be replaced by a sequence of queries to all neighbors of v, obtaining not less information at strictly smaller cost. In general, given an instance which does not satisfy the star condition, we enforce it by performing all necessary weight replacements $w(v) \leftarrow \min\{w(v), \sum_{u \in N(v)} w(u)\}$, for $v \in V$.

For $a, \omega \in \mathbb{R}_{\geq 0}$, we denote the rounding of a down (up) to the nearest multiple of ω as $\lfloor a \rfloor_\omega = \omega \lfloor a/\omega \rfloor$ and $\lceil a \rceil_\omega = \omega \lceil a/\omega \rceil$, respectively.

3.2.2 Definition of a Search Strategy

A search strategy $\mathcal{A}$ for a rooted tree $T = (V, E, w)$ is an adaptive algorithm which defines successive queries to the tree, based on responses to previous queries, with the objective of locating the target vertex in a finite number of steps. Note that search strategies can be seen as decision trees in which each node represents a subset of vertices of T that contains x, with leaves representing singletons consisting of x.

Let $\mathtt{Q}_{\mathcal{A}}(T, x)$ be the time-ordering (sequence) of queries performed by strategy $\mathcal{A}$ on tree T to find a target vertex x, with $\mathtt{Q}_{\mathcal{A},i}(T, x)$ denoting the i-th queried vertex in this time ordering, $1 \leq i \leq |\mathtt{Q}_{\mathcal{A}}(T, x)|$.

We denote by

$$\mathtt{COST}_{\mathcal{A}}(T, x) = \sum_{i=1}^{|\mathtt{Q}_{\mathcal{A}}(T,x)|} w(\mathtt{Q}_{\mathcal{A},i}(T, x))$$

the sum of weights of all vertices queried by $\mathcal{A}$ with x being the target node, i.e., the time after which $\mathcal{A}$ finishes. Let

$$\mathtt{COST}_{\mathcal{A}}(T) = \max_{x \in V} \mathtt{COST}_{\mathcal{A}}(T, x)$$

be the *cost of* $\mathcal{A}$. We define the *cost of* T to be

$$\mathtt{OPT}(T) = \min\{\mathtt{COST}_{\mathcal{A}}(T) \;\big|\; \mathcal{A} \text{ is a search strategy for } T\}.$$

We say that a search strategy is *optimal* for T if its cost equals $\mathtt{OPT}(T)$.

As a consequence of normalization and the star condition, we have the following bound.

Observation 3.2.1. *For any tree T, we have $1 \leq \mathtt{OPT}(T) \leq \lceil \log_2 n \rceil$.*

Proof. By the star condition, considering any vertex $v \in V$ as the target, we trivially have

$$\mathtt{OPT}(T) \geq \inf_{\mathcal{A}} \mathtt{COST}_{\mathcal{A}}(T, v) \geq \inf_{\mathcal{A}} \mathtt{COST}_{\mathcal{A}}(T[\{v\} \cup N(v)], v) \geq w(v).$$

Thus, $\texttt{OPT}(T) \geq \max_{v \in V} w(v) = 1$, which gives the first inequality.

For the second inequality, we observe that applying to tree T the optimal search strategy for unweighted trees, we can locate the target in at most $\lceil \log_2 n \rceil$ queries (cf. e.g. [54, 76]). Since the cost of each query is at most 1, the claim follows. □

We also introduce the following notation. If the first $|U|$ queried vertices by a search strategy $\mathcal{A}$ are exactly the vertices in U, $U = \{\texttt{Q}_{\mathcal{A},i}(T, x) : 1 \leq i \leq |U|\}$, then we say that $\mathcal{A}$ *reaches* $T\langle U, x\rangle$ *through* U, and $w(U)$ is the *cost of reaching* $T\langle U, x\rangle$ *by* $\mathcal{A}$. We also say that we receive an 'up' reply to a query to a vertex v if the root of the tree remaining to be searched remains unchanged by the query, i.e., $r(T\langle U, x\rangle) = r(T\langle U \cup \{v\}, x\rangle)$, and we call the reply a 'down' reply when the root of the remaining tree changes, i.e., $r(T\langle U, x\rangle) \neq r(T\langle U \cup \{v\}, x\rangle)$. Without loss of generality, after having performed a sequence of queries U, we can assume that the tree $T\langle U, x\rangle$ is known to the strategy.

3.3 Valid Strategies and its Characterization

3.3.1 Presentation of Valid Strategies

We call a search strategy *polynomial-time* if it can be implemented using a dynamic (adaptive) algorithm which computes the next queried vertex in polynomial time.

We give most of our attention herein to search strategies in trees which admit a natural (non-adaptive, polynomial-space) representation called a *query sequence assignment*. Formally, for a rooted tree T, the *query sequence assignment* S is a function $S : V \to V^*$, which assigns to each vertex $v \in V$ an ordered sequence of vertices $S(v)$, known as the *query sequence* of v. The query sequence assignment directly induces a strategy $\mathcal{A}_S$, presented as Algorithm 4. Intuitively, the strategy processes successive queries from the sequence $S(v)$, where v is the root vertex of the current search tree, $v = r(T\langle U, x\rangle)$, where U is the set of queries performed so far. This processing is performed in such a way that the strategy iteratively takes the first vertex in $S(v)$ that belongs to $T\langle U, x\rangle$ and queries it. As soon as the root of the search tree changes, the procedure starts processing queries from the query sequence of the new root, which belong to the remaining search tree. The procedure terminates as soon as $T\langle U, x\rangle$ has been reduced to a single vertex, which is necessarily the target x.

In what follows, in order to show that our approximation strategies are polynomial-time, we will confine ourselves to presenting a polynomial-time algorithm which outputs an appropriate sequence assignment.

A sequence assignment is called *stable* if the replacement of line 9 in Algorithm 4 by any assignment of the form $v \leftarrow v''$, where v'' is an arbitrary vertex which is promised to lie on the path from $r(T\langle U, x\rangle)$ to the target x, always results in a strategy which performs a (not necessarily strict) subsequence of the sequence of queries performed by the original strategy $\mathcal{A}_S$. Sequence assignments computed on trees with a bottom-up approach usually have the stability property; we provide a proof of stability for one of our main routines in Section 3.4.3.

Algorithm 4 Search strategy $\mathcal{A}_S$ for a query sequence assignment S

```
1: v ← r(T)    // stores current root
2: U ← ∅
3: while |T⟨U, x⟩| > 1 do
4:     for u ∈ S(v) do
5:         if u ∈ T⟨U, x⟩ then    // u is the first vertex in S(v) that belongs to T⟨U, x⟩
6:             QueryVertex(u)
7:             U ← U ∪ {u}
8:             if v ≠ r(T⟨U, x⟩) then    // query reply is 'down'
9:                 v ← r(T⟨U, x⟩)
10:                break    // for loop
11:            end if
12:        end if
13:    end for
14: end while
```

Without loss of generality, we will also assume that if $v \in S(v)$, then v is the last element of $S(v)$. Indeed, when considering a subtree rooted at v, after a query to v, if v was not the target, then the root of the considered subtree will change to one of the children of v, hence any subsequent elements of $S(v)$ may be removed without changing the strategy.

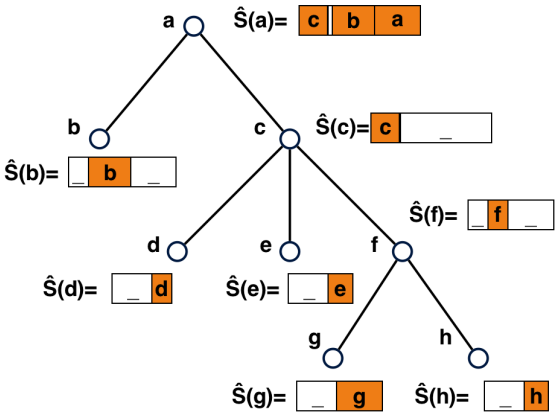
3.3.2 Strategies Based on Consistent Schedules

Intuitively, we may represent search strategies by a schedule consisting of some number of jobs, with each job being associated to querying a node in the tree (cf. e.g. [51, 65, 66]). Each job has a fixed processing time, which is set to the weight of a node. Formally, in this work we will refer to the schedule $\hat{S}$ only in the very precise context of search strategies $\mathcal{A}_S$ based on some query sequence assignment S. The *schedule assignment* $\hat{S}$ is the following extension of the sequence assignment S, which additionally encodes the starting time of search query job. If the query sequence S of a node v is of the form $S(v) = (v_1, \ldots, v_k)$, $k = |S(v)|$, then the corresponding schedule for v will be given as $\hat{S}(v) = ((v_1, t_1), \ldots, (v_k, t_k))$, with $t_i \in \mathbb{R}_{\geq 0}$ denoting the starting time of the query for v_i. We will call $\hat{S}(v)$ the *schedule of node* v. We will call a schedule assignment $\hat{S}$ *consistent* with respect to search in a given tree T if the following conditions are fulfilled:

(i) No two jobs in the schedule of a node overlap: for all $v \in V$, for two distinct jobs $(u_1, t_1), (u_2, t_2) \in \hat{S}(v)$, we have $|[t_1, t_1 + w(u_1)] \cap [t_2, t_2 + w(u_2)]| = 0$. Here $|[a, b]|$ is the length of the interval, i.e. $|[a, b]| = b - a$.

(ii) If v is the parent of v' in T and $(u, t) \in \hat{S}(v')$, then we either also have $(u, t) \in \hat{S}(v)$, or the job $(v, t_v) \in \hat{S}(v)$ completes before the start of job (u, t): $t_v + w(v) \leq t$.

It follows directly from the definition that a consistent schedule assignment (and the underlying query sequence assignment) is uniquely determined by the collection of



search for b:
Query c (the first item in $\hat{S}(a)$); Reply: a (by oracle)
Query b (the second item of $\hat{S}(a)$, since replied a;) Reply: yes.
search for h:
Query c (the first item in $\hat{S}(a)$); Reply: f
Query f (the first item in $\hat{S}(f)$); Reply: h
Query h (the first item in $\hat{S}(h)$); Reply: yes.

Figure 3.1: a schedule assignment represented by intervals and two examples of searching for b and h.

jobs $\{(v, t_v) : (v, t_v) \in \hat{S}(u), u \in V\}$. Note that not every vertex has to contain a query to itself in its schedule; we will occasionally write $t_v = \perp$ to denote that such a job is missing. In this case, the jobs of all children of v have to be contained in the schedule of node v.

By extension of notation for sequence assignments, we will denote a strategy following a consistent schedule assignment $\hat{S}$ (i.e., executing the query jobs of schedule $\hat{S}$ at the prescribed times) as $\mathcal{A}_{\hat{S}}$. We will then have:

$$\texttt{COST}_{\mathcal{A}_{\hat{S}}}(T) = |\hat{S}|,$$

where $|\hat{S}|$ is the *duration* of schedule assignment $\hat{S}$, given as:

$$|\hat{S}| = \max_{v \in V} |\hat{S}(v)|,$$

with:

$$|\hat{S}(v)| = \max_{(u,t) \in \hat{S}(v)} (t + w(u)).$$

We remark that there always exists an optimal search strategy which is based on a consistent schedule. By a well-known characterization (cf. e.g. [28]), tree T satisfies $\texttt{OPT}(T) = \tau \in \mathbb{R}$ if and only if there exists an assignment $I : V \to \mathcal{I}_\tau$ of intervals of time to nodes before deadline τ, $\mathcal{I}_\tau = \{[a, b] : 0 \leq a < b \leq \tau\}$, such that $|I(v)| = w(v)$ and if $|I(u) \cap I(v)| > 0$ for any pair of nodes $u, v \in V$, then the $u - v$ path in T contains a separating vertex z such that $\max I(z) \leq \min(I(u) \cup I(v))$. The corresponding schedule assignment of duration τ is obtained by adding, for each

node $u \in V$, the job $(u, \min I(u))$ to the schedule of all nodes on the path from u towards the root, until a node v such that $\max I(v) \leq \min I(u)$ is encountered on this path. The consistency and correctness of the obtained schedule is immediate to verify.

Observation 3.3.1. *For any tree T, there exists a query sequence assignment S and a corresponding consistent schedule $\hat{S}$ on T such that $|\hat{S}| = \mathtt{OPT}(T)$.*

3.4 $(1+\varepsilon)$-Approximation in $n^{O(\log n/\varepsilon^2)}$ Time

We at first present an approximation scheme for the weighted tree search problem with $n^{O(\log n)}$ running time. The main difficulty consists in obtaining a constant approximation ratio for the problem with this running time; we at once present this approximation scheme with tuned parameters, so as to achieve $(1+\varepsilon)$-approximation in $n^{O(\log n/\varepsilon^2)}$ time.

Our construction consists of two main building blocks. First, we design an algorithm based on a bottom-up (dynamic programming) approach, which considers exhaustively feasible sequence assignments and query schedules over a carefully restricted state space of size $n^{O(\log n)}$ for each node. The output of the algorithm provides us both with a lower bound on $\mathtt{OPT}(T)$, and with a sequence assignment-based strategy $\mathcal{A}_S$ for solving the tree search problem. The performance of this strategy $\mathcal{A}_S$ is closely linked to the performance of $\mathtt{OPT}(T)$, however, there is one type of query, namely a query on a vertex of small weight leading to a 'down' response, due to whose repeated occurrence the eventual cost difference between $\mathtt{COST}_{\mathcal{A}_S}(T)$ and $\mathtt{OPT}(T)$ may eventually become arbitrarily large. To alleviate this difficulty, we introduce an alternative measure of cost which compensates for the appearance of the disadvantageous type of query.

3.4.1 Modified Costs

We start by introducing some additional notation. Let $\omega \in \mathbb{R}_+$, be an arbitrarily fixed value of weight and let $c \in \mathbb{N}$. The choice of constant $c \in \mathbb{N}$ will correspond to an approximation ratio of $(1+\varepsilon)$ of the designed scheme for $\varepsilon = 168/c$.

We say that a query to a vertex v is a *light down query* in some strategy $\mathcal{A}$ if $w(v) < c\omega$ and $x \in V(T_v)$, i.e., it is also a 'down' query, where x is the target vertex.

For any strategy $\mathcal{A}$, we denote by $\mathtt{COST}^{(\omega,c)}_{\mathcal{A}}(T,x)$ its modified cost of finding target x, defined as follows. Let d_x be the number of light down queries when searching for x:

$$d_x = \left|\{i : w(\mathtt{Q}_{\mathcal{A},i}(T,x)) < c\omega \text{ and } x \in V(T_{\mathtt{Q}_{\mathcal{A},i}(T,x)})\}\right|.$$

Then, the modified cost $\mathtt{COST}^{(\omega,c)}_{\mathcal{A}}(T,x)$ is:

$$\mathtt{COST}^{(\omega,c)}_{\mathcal{A}}(T,x) = \mathtt{COST}_{\mathcal{A}}(T,x) - (2c+1)\omega d_x. \tag{3.1}$$

and by a natural extension of notation:

$$\mathtt{COST}^{(\omega,c)}_{\mathcal{A}}(T) = \max_{x \in V} \mathtt{COST}^{(\omega,c)}_{\mathcal{A}}(T,x).$$

The technical result which we will obtain in Subsection 3.4.3 may now be stated as follows.

Proposition 3.4.1. *For any $c \in \mathbb{N}$, $L \in \mathbb{N}$, there exists an algorithm running in time $(cn)^{O(L)}$, which for any tree T constructs a stable sequence assignment S and computes a value of ω such that $\omega \leq \frac{1}{L}\mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T)$ and:*

$$\mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) \leq \left(1 + \frac{12}{c}\right)\mathtt{OPT}(T).$$

In order to convert the obtained strategy $\mathcal{A}_S$ with a small value of $\mathtt{COST}^{(\omega,c)}$ into a strategy with small $\mathtt{COST}$, we describe in Section 3.4.5 an appropriate strategy conversion mechanism. The approach we adopt is applicable to any strategy based on a stable sequence assignment and consists in concatenating, for each vertex $v \in V$, a prefix to the query sequence $S(v)$ in the form of a separately computed sequence $R(v)$, which does not depend on $S(v)$. The considered query sequences are thus of the form $R(v) \circ S(v)$, where the symbol "$\circ$" represents sequence concatenation. Intuitively, the sequences R, taken over the whole tree, reflect the structure of a specific solution to the unweighted tree search problem on a contraction of tree T, in which each edge connecting a node to a child with weight at least $c\omega$ is contracted. We recall that the optimal number of queries to reach a target in an unweighted tree is $O(\log n)$, and the goal of this conversion is to reduce the number of light down queries in the combined strategy to at most $O(\log n)$.

Proposition 3.4.2. *For any fixed $\omega > 0$ there exists a polynomial-time algorithm which for a tree T computes a sequence assignment $R : V \to V^*$, such that, for any strategy $\mathcal{A}_S$ based on a stable sequence assignment S, the sequence assignment S^+, given by $S^+(v) = R(v) \circ S(v)$ for each $v \in V$, has the following property:*

$$\mathtt{COST}_{\mathcal{A}_{S^+}}(T) \leq \mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) + 4(2c+1)\omega \log_2 n.$$

The proof of Proposition 3.4.2 is provided in Section 3.4.5.

We are now ready to put together the two bounds. Combining the claims of Proposition 3.4.1 for $L = \lceil c^2 \log_2 n \rceil$ (with $\omega \leq \frac{1}{L}\mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) \leq \frac{\mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T)}{c^2 \log_2 n}$) and Proposition 3.4.2, we obtain:

$$\begin{aligned}
\mathtt{COST}_{\mathcal{A}_{S^+}}(T) &\leq \mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) + 4(2c+1)\omega \log_2 n \leq \mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) + 12c\omega \log_2 n \;\leq \\
&\leq \mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) + 12c \log_2 n \frac{\mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T)}{c^2 \log_2 n} \leq \left(1 + \frac{12}{c}\right)\mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) \leq \\
&\leq \left(1 + \frac{12}{c}\right)^2 \mathtt{OPT}(T) \leq \left(1 + \frac{168}{c}\right)\mathtt{OPT}(T).
\end{aligned}$$

After putting $\varepsilon = \frac{168}{c}$ and noting that in stating our result we can safely assume $c = O(\mathrm{poly}(n))$ (beyond this, the tree search problem can be trivially solved optimally in $O(n^n)$ time using exhaustive search), we obtain the main theorem of the Section.

Theorem 3.4.1. *There exists an algorithm running in $n^{O\left(\frac{\log n}{\varepsilon^2}\right)}$ time, providing a $(1+\varepsilon)$-approximation solution to the weighted tree search problem.*

3.4.2 Preprocessing: Time Alignment in Schedules

We adopt here a method similar but arguably more refined than rounding techniques in scheduling problems of combinatorial optimization, showing that we could discretise the starting and finishing time of jobs, as well as weights of vertices, in a way to restrict the size of state space for each node to $n^{O(\log n)}$, without introducing much error.

Fix $c \in \mathbb{N}$ and $\omega = \frac{a}{cn}$ for some $a \in \mathbb{N}$. (In subsequent considerations, we will have $c = \Theta(1/\varepsilon)$, $a = O(\frac{n}{\log n})$ and $\omega = \Omega(\varepsilon/\log n)$.) Given a tree $T = (V, E, w)$, let $T' = (V, E, w')$ be a tree with the same topology as T but with weights rounded up as follows:

$$w'(v) = \begin{cases} \lceil w(v) \rceil_{\omega}, & \text{if } w(v) > c\omega, \\ \lceil w(v) \rceil_{\frac{1}{cn}}, & \text{otherwise.} \end{cases} \tag{3.2}$$

We will informally refer to vertices with $w(v) > c\omega$ (equivalently $w'(v) > c\omega$) as *heavy vertices* and vertices with $w(v) \leq c\omega$ (equivalently $w'(v) \leq c\omega$) as *light vertices*. (Note that $w(v) \leq c\omega$ if and only if $w'(v) \leq c\omega$.) When designing schedules, we consider time divided into *boxes* of duration ω, with the i-th box equal to $[i\omega, (i+1)\omega]$. Each box is divided into a identical *slots* of length $\frac{1}{cn}$.

In the tree T', the duration of a query to a heavy vertex is an integer number of boxes, and the duration of a query to a light vertex is an integer number of slots. We next show that, without affecting significantly the approximation ratio of the strategy, we can align each query to a heavy vertex in the schedule so that it occupies an interval of full adjacent boxes, and each query to a light vertex in the schedule so that it occupies an interval of full adjacent slots (possibly contained in more than one box).

We start by showing the relationship between the costs of optimal solutions for trees T and T'.

Lemma 3.4.1. $\mathtt{OPT}(T) \leq \mathtt{OPT}(T') \leq (1 + \frac{2}{c})\mathtt{OPT}(T)$.

Proof. The inequality $\mathtt{OPT}(T) \leq \mathtt{OPT}(T')$ follows directly from the monotonicity of the cost of the solution with respect to vertex weights, since we have $w'(v) \geq w(v)$, for all $v \in V$.

To show the second inequality, we note that by the definition of weights (3.2), for any vertex v, $w'(v) \leq (1 + \frac{1}{c})w(v) + \frac{1}{cn}$.

Consider an optimal strategy $\mathcal{O}$ for tree T and let $\mathtt{Q}_{\mathcal{O}}(T, x) = (v_1, \ldots, v_k)$ be the time-ordering of queries performed by strategy $\mathcal{O}$ on tree T to find a target vertex x. Let $\mathcal{O}'$ be the strategy which follows the same time-ordering of queries when locating target x in T'. We have:

$$\begin{aligned} \mathtt{COST}_{\mathcal{O}'}(T', x) = \sum_{i=1}^{k} w'(v_i) \leq \sum_{i=1}^{k} \left(\left(1 + \frac{1}{c}\right) w(v) + \frac{1}{cn} \right) \leq \frac{1}{c} + \left(1 + \frac{1}{c}\right) \sum_{i=1}^{k} w(v) \leq \\ \leq \left(1 + \frac{2}{c}\right) \mathtt{OPT}(T), \end{aligned}$$

where we used the fact that, by Observation 3.2.1, $\mathtt{OPT}(T) \geq 1$. Since $\mathtt{OPT}(T') \leq \max_{x \in V} \mathtt{COST}_{\mathcal{O}'}(T', x)$, the claim follows. □

Lemma 3.4.2. *There exists a consistent schedule assignment $\hat{S}$ for tree T' such that $\texttt{COST}_{\mathcal{A}_{\hat{S}}}(T') \leq (1+\frac{3}{c})\texttt{OPT}(T')$ and for all $v \in V$ we have that*

- *if $w'(v) > c\omega$, (v is heavy), then the starting time t of any job (v,t) in the schedule $\hat{S}(u)$ of any $u \in V$ is an integer multiple of ω (aligned to a box),*
- *if $w'(v) \leq c\omega$, (v is light), then the starting time t of any query (v,t) in the schedule $\hat{S}(u)$ of any $u \in V$ is an integer multiple of $\frac{1}{cn}$ (aligned to a slot).*

Proof. We consider an optimal consistent schedule assignment $\hat{\Sigma}$ for tree T', $|\hat{\Sigma}| = \texttt{OPT}(T')$. Fix $u \in V$ arbitrarily, and let $(v_{u,i}, t_{u,i})$ be the i-th query job in $\hat{\Sigma}(u)$. Consider now the schedule $\hat{\Sigma}^*(u)$ for T based on the same sequence assignment, in which the job $(v_{u,i}, t_{u,i})$ is replaced by the job $(v_{u,i}, t^*_{u,i})$ with $t^*_{u,i} = (1+\frac{2}{c})t_{u,i}$. We have for any two consecutive jobs at u:

$$t^*_{u,i+1} - t^*_{u,i} = \left(1+\frac{2}{c}\right)(t_{u,i+1} - t_{u,i}) \geq \left(1+\frac{2}{c}\right) w(v_{u,i}), \tag{3.3}$$

where we assume by convention that for the last job index i_{max}, $t_{u,i_{max}+1} = |\hat{\Sigma}(u)|$. We now observe that schedule assignment $\hat{\Sigma}^*$ on tree T can be directly converted into schedule assignment $\hat{S}$ on tree T' as follows. The query sequence of each vertex is preserved unchanged. If $v_{u,i}$ is a heavy vertex, then within time interval $[t^*_{u,i}, t^*_{u,i+1}]$ we allocate to vertex $v_{u,i}$ an interval of full boxes, starting at time $\lceil t^*_{u,i} \rceil_\omega$. Indeed, by (3.3) we have:

$$t^*_{u,i+1} - \lceil t^*_{u,i} \rceil_\omega > t^*_{u,i+1} - t^*_{u,i} - \omega > \left(1+\frac{2}{c}\right) w(v_{u,i}) - \omega > w(v_{u,i}) + \omega > w'(v_{u,i}).$$

Since no two jobs overlap and the time transformation is performed identically for all vertices, the validity and consistency of schedule assignment $\hat{S}$ for tree T' follows. We also have $|\hat{S}| \leq (1+\frac{2}{c})|\hat{\Sigma}| = (1+\frac{2}{c})\texttt{OPT}(T')$.

To obtain the second part of the claim (alignment for light vertices) it suffices to round up the starting time of query times of all (light) vertices to an integer multiple of $\frac{1}{cn}$. Since all weights in T' are integer multiples of $\frac{1}{cn}$, and so are the starting times of queries to heavy vertices in $\hat{S}$, the correctness and consistency of the obtained schedule again follows directly. This final transformation increases the duration by at most $\frac{1}{c} \leq \frac{1}{c}\texttt{OPT}(T')$, and combining the bounds for both the transformations finally gives the claim. □

A schedule on tree T' satisfying the conditions of Lemma 3.4.2, and the resulting search strategy, are called *aligned*. Subsequently, we will design an aligned strategy on tree T', and compare the quality of the obtained solution to the best aligned strategy for T'.

The intuition between the separate treatment of heavy vertices (aligned to boxes) and light vertices (aligned to slots) in aligned schedules is the following. Whereas the time ordering of boxes is essential in the design of the correct strategy, in our dynamic programming approach we will not be concerned about the order of slots within a single box (i.e., the order of queries to light vertices placed in a single box).

This allows us to reduce the state space of a node. Whereas the ordering of slots in the box will eventually have to be repaired to provide a correct strategy, this will not affect the quality of the overall solution too much (except for the issue of light down queries pointed out earlier, which are handled separately in Section 3.4.5).

3.4.3 Dynamic Programming Routine for Fixed Box Size

Let the values of parameter c and box size ω be fixed as before. Additionally, let $L \in \mathbb{N}$ be a parameter representing the time limit for the duration of the considered vertex schedules when measured in boxes, i.e., the longest schedule considered by the procedure will be of length $L\omega$ (we will eventually choose an appropriate value of $L = O(\log n)$ as required when showing Theorem 3.4.1).

Before presenting formally the considered quasi-polynomial time procedure, we start by outlining an (exponential time) algorithm which verifies if there exists an aligned schedule assignment $\hat{\Sigma}$ for T' whose duration is at most $L\omega$. Notice that since all weights in T' are integer multiples of $\frac{1}{cn}$, the optimal aligned schedule assignment will start and complete the execution of all queries at times which are integer multiples of $\frac{1}{cn}$; thus, we may restrict the considered class of schedules to those having this property. Any possible schedule of length at most $L\omega$ at a vertex v, which may appear in $\hat{\Sigma}$, will be represented in the form of the pair (σ_v, t_v), where:

- σ_v is a Boolean array with $L\omega cn$ entries, where $\sigma_v[i] = 1$ when time slot $[\frac{i}{cn}, \frac{i+1}{cn}]$ is occupied in the schedule at v, and $\sigma_v[i] = 0$ otherwise.
- $t_v \in \mathbb{R}$ represents the start time of the query to v in the schedule of v (we put $t = \perp$ if such a query does not appear in the schedule).

We now state some necessary conditions for a consistent schedule, known from the analysis of the unweighted search problem (cf. e.g. [50, 76, 81]). The first observation expresses formally the constraint that the same time slot cannot be used in the schedules of two children of a node v, unless it is separated by an (earlier) query to node v itself. All time slots before the starting time t_v of job (v, t_v) are free if and only if the corresponding time slot is free for all of the children of v.

Observation 3.4.1. *Assume that the tuple $(\sigma_v, t_v)_{v \in V}$ corresponds to a consistent schedule. Let $v \in V$ be an arbitrarily chosen node with set of children $\{v_1, \ldots, v_l\}$. Let the completion time t^v_{end} of the query to v in the schedule of v be given as:*

$$t^v_{end} = \begin{cases} t_v + w'(v), & \text{if } t_v \neq \perp, \\ +\infty, & \text{if } t_v = \perp. \end{cases}$$

Then, for any time slot $[\frac{i}{cn}, \frac{i+1}{cn}]$, we have:

$$\left.\begin{aligned} &\sigma_v[i] = \sum_{j=1}^{l} \sigma_{v_j}[i], && \text{when } \frac{i+1}{cn} \leq t_v, \\ &\sigma_v[i] = 1 \text{ and } \sum_{j=1}^{l} \sigma_{v_j}[i] = 0, && \text{when } t_v < \frac{i+1}{cn} \leq t^v_{end}, \\ &\sigma_v[i] = 0, && \text{when } \frac{i+1}{cn} > t^v_{end}. \end{aligned}\right\} \tag{3.4}$$

We remark that the last of the above conditions (3.4) follows from the w.l.o.g. assumption we made when defining sequence assignments that whenever node v appears in the schedule of v, it is the last node in the query sequence for v.

Moreover, any valid search strategy which locates a target vertex must eventually query at least one of the endpoints of every edge of the tree T', since otherwise, it will not be able to distinguish targets located at these two endpoints. We thus make the following observation.

Observation 3.4.2. *Assume that the tuple $(\sigma_v, t_v)_{v\in V}$ represents a consistent schedule. Let $v \in V$ be an arbitrarily chosen node with set of children $\{v_1, \ldots, v_l\}$. Then:*

$$\text{If } t_v = \perp, \text{ then } t_{v_j} \neq \perp, \text{ for all } 1 \leq j \leq l. \tag{3.5}$$

Conditions (3.4) and (3.5) provide us with necessary conditions which must be satisfied by any consistent aligned schedule assignment.

In order to lower-bound the duration of the consistent aligned schedule assignment with minimum cost, we perform an exhaustive bottom-up evaluation of aligned schedules which satisfy the constraints of (3.5), and a slightly weaker form of the constraints of (3.4). These weaker constraints are introduced to reduce the running time of the algorithm. Instead of considering individual slots of a schedule which may be empty or full, $\sigma_v[i] \in \{0, 1\}$, we consider the load of each box in the same schedule, defined as the proportion of occupied slots within the box. Formally, for the p-th box, $0 \leq p < L$, the *load* $s_v[p]$ is given as:

$$s_v[p] = \frac{1}{\omega cn} \sum_{i=p\cdot\omega cn}^{(p+1)\omega cn-1} \sigma_v[i], \qquad s_v[p] \in \left\{0, \frac{1}{\omega cn}, \frac{2}{\omega cn}, \ldots, 1\right\},$$

where we recall that ωcn is an integer by the choice of ω. We will call a box with load $s_v[p] = 0$ an *empty box*, a box with load $s_v[p] = 1$ a *full box*, and a box with load $0 < s_v[p] < 1$ a *partially full box* in the schedule of v.

By summing over all slots within each box, we obtain the following corollary directly from Observation 3.4.1.

Corollary 3.4.1. *Assume that the tuple $(s_v, t_v)_{v\in V}$ corresponds to a consistent schedule. Let $v \in V$ be an arbitrarily chosen node with set of children $\{v_1, \ldots, v_l\}$ and completion time t^v_{end} of the query to v given as in Observation 3.4.1. Let a_p be the contribution to the load of the p-th box of the query job for vertex v, i.e.*

$$a_p = \begin{cases} \frac{1}{\omega}|[t_v, t^v_{end}] \cap [p\omega, (p+1)\omega]| & \text{if } t_v \neq \perp, \\ 0 & \text{if } t_v = \perp. \end{cases}$$

Then, for any box $[p\omega, (p+1)\omega]$, $0 \leq p < L$, we have:

$$\left.\begin{array}{ll} s_v[p] = a_p + \sum_{j=1}^{l} s_{v_j}[p] \in [0, 1], & \text{when } t^v_{end} \geq (p+1)\omega, \\ s_v[p] \geq a_p, & \text{when } p\omega < t^v_{end} < (p+1)\omega, \\ s_v[p] = 0, & \text{when } t^v_{end} \leq p\omega. \end{array}\right\} \tag{3.6}$$

Moreover, for any box $[p\omega, (p+1)\omega]$, $0 \leq p < L$, *we have:*

$$\text{For all } 1 \leq j \leq l, \text{ the following bound holds: } s_{v_j}[p] + a_p \leq 1. \tag{3.7}$$

We remark that the statement of Corollary 3.4.1 treats specially one box, namely the one which contains strictly within it the time moment t^v_{end}. For this box, we are unable to make a precise statement about $s_v[p]$ based on the description of the schedules of its children, and content ourselves with a (potentially) weak lower bound $s_v[p] \geq a_p = \frac{1}{\omega}(t^v_{end} - p\omega)$. This is the direct reason for the slackness in our subsequent estimation, which loses ω time per down query. However, we note that by the definition of aligned schedule, a query to a heavy vertex will never begin or end strictly inside a box, and will not lead to the appearance of this issue. We remark that condition (3.7) additionally stipulates that within any box, it must be possible to schedule the contribution of the query to v and the contribution of any child v_j to the load of the box in a non-overlapping way.

We now show that the shortest schedule assignments satisfying the set of constraints (3.5), (3.6), and (3.7) can be found in $n^{O(\log n)}$ time. This is achieved using a procedure BuildStrategy, presented in Algorithm 5, which returns for a node v a non-empty set of schedules $\hat{\mathcal{S}}[v]$, such that each $s_v \in \hat{\mathcal{S}}[v]$ can be extended into the sought assignment of schedules in its subtree, $(s_u, t_u)_{u \in V(T_v)}$. In the statement of Algorithm 5, we recall that, given a tree $T = (V, E, w)$, tree $T' = (V, E, w')$ is the tree with weights rounded up to the nearest multiple of the length of a slot (see Equation (3.2)).

The subsequent steps taken in procedure BuildStrategy can be informally sketched as follows. The input tree T' is processed in a bottom-up manner and hence, for an input vertex v, the recursive calls for its children $v_1, \ldots, v_l$ are first made, providing schedule assignments for the children (see lines 3-4). Then, the rest of the pseudocode is responsible for using these schedule assignments to obtain all valid schedule assignments for v. Lines 11-15 merge the schedules of the children in such a way that a set $\hat{S}^*_i$, $i \in \{1, \ldots, l\}$, contains all schedule assignments computed on the basis of the schedules for the children $v_1, \ldots, v_i$. Thus, the set $\hat{S}^*_l$ is the final product of this part of the procedure and is used in the remaining part. Note that a schedule assignment in $\hat{S}^*_l$ may not be valid since a query to v is not accommodated in it — the rest of the pseudocode is responsible for taking each schedule $s \in \hat{S}^*_l$ and inserting a query to v into s. More precisely, the subroutine InsertVertex is used to place the query to v at all possible time points (depending whether v is heavy or light). We note that the subroutine MergeSchedules, for each schedule s it produces, sets a Boolean 'flag' $s.must\_contain\_v$ that whenever equals $false$, indicates that querying v is not necessary in s to obtain a valid schedule for v (this happens if s queries all children of v). A detailed analysis of procedure BuildStrategy can be found in the proof of Lemma 3.4.3.

Lemma 3.4.3. *For fixed constants* $L, c \in \mathbb{N}$, *calling procedure* BuildStrategy$(r(T), \omega)$, *where* $r(T)$ *is the root of the tree, determines if there exists a tuple* $(s_v, t_v)_{v \in V}$ *which satisfies constraints* (3.5), (3.6), *and* (3.7), *or returns an empty set otherwise.*

Proof. The formulation of procedure BuildStrategy directly enforces that the constraints (3.5), (3.6), and (3.7) are fulfilled at each level of the tree, in a bottom-up manner.

Algorithm 5 Dynamic programming routine BUILDSTRATEGY for a tree T'. $L, c \in \mathbb{N}$ are global parameters. Subroutines MERGESCHEDULES and INSERTVERTEX are provided further on.

procedure BUILDSTRATEGY(vertex v, box size $\omega \in \mathbb{R}$)
  $l \leftarrow$ number of children of v in T' // Denote by $v_1, \ldots, v_l$ the children of v.
  for $i = 1..l$ **do**
    $\hat{\mathcal{S}}[v_i] \leftarrow$ BUILDSTRATEGY(v_i, ω);
  end for
  $s \leftarrow 0^L$
  $s.max\_child\_load \leftarrow 0^L$
  $s.must\_contain\_v \leftarrow false$
  $\hat{\mathcal{S}}_0 \leftarrow \{s\}$ // $\hat{\mathcal{S}}_0$ contains the schedule with no queries.
  // Inductively, $\hat{\mathcal{S}}_i^*$ is based on merging schedules at $v_1, \ldots, v_i$.
  for $i = 1..l$ **do**
    $\hat{\mathcal{S}}_i^* \leftarrow \emptyset$
    for each schedule $s \in \hat{\mathcal{S}}_{i-1}^*$ **do**
      for each schedule $s_{add} \in \hat{\mathcal{S}}[v_i]$ **do**
        $\hat{\mathcal{S}}_i^* \leftarrow \hat{\mathcal{S}}_i^* \cup$ MERGESCHEDULES(s, s_{add}, ω);
      end for
    end for
  end for
  $\hat{\mathcal{S}}[v] \leftarrow \emptyset$
  for each $s \in \hat{\mathcal{S}}_l^*$ **do**
    if $w'(v) > c\omega$ **then** // v is heavy
      for $p = 0..L-1$ **do** //attempt to insert (into s) query to v starting from time-box p
        $\hat{\mathcal{S}}[v] \leftarrow \hat{\mathcal{S}}[v] \cup$ INSERTVERTEX($s, v, \omega, p \cdot \omega$)
      end for
    else //v is light
      for real $t = 0..L \cdot \omega$ **step** $\frac{1}{cn}$ **do**
        //attempt to insert (into s) query to v at a slot from time t
        $\hat{\mathcal{S}}[v] \leftarrow \hat{\mathcal{S}}[v] \cup$ INSERTVERTEX(s, v, ω, t)
      end for
    end if
    if $s.must\_contain\_v = false$ **then**
      $\hat{\mathcal{S}}[v] \leftarrow \hat{\mathcal{S}}[v] \cup$ INSERTVERTEX($s, v, \omega, \perp$)
    end if
  end for
  return $\hat{\mathcal{S}}[v]$
end procedure

Algorithm 6 Subroutines MergeSchedules and InsertVertex of procedure BuildStrategy from Algorithm 5.

procedure MergeSchedules(schedule s_{orig}, schedule s_{add}, box size $\omega \in \mathbb{R}$)
  $s \leftarrow s_{orig}$ // copy schedule and its properties to answer
  for $p = 0..L-1$ **do** // for each time-box add load of s_1 and s_2
    $s[p] \leftarrow s_{orig}[p] + s_{add}[p]$
    if $s[p] > 1$ **then**
      $s[p] \leftarrow +\infty$
    end if
    $s.max\_child\_load[p] \leftarrow \max\{s.max\_child\_load[p], s_{add}[p]\}$
  end for
  if $s_{add}.t_v = \perp$ **then**
    $s.must\_contain\_v \leftarrow true$
  end if
  return s
end procedure

procedure InsertVertex(schedule s_{orig}, vertex v, box size $\omega \in \mathbb{R}$, time $t \in \mathbb{R} \cup \{\perp\}$)
  $s \leftarrow 0^L$ // initialize empty schedule for answer
  if $t \neq \perp$ **then**
    $I \leftarrow [t, t + w'(v)]$ // time interval into which query to v is being inserted
    $s.t_v \leftarrow t$
    $t^v_{end} \leftarrow t + w'(v)$
  else
    $I \leftarrow \emptyset$
    $s.t_v \leftarrow \perp$
    $t^v_{end} \leftarrow +\infty$
  end if
  for $p = 0..L-1$ **do** // for each time-box
    $a_p \leftarrow \frac{1}{\omega}|I \cap [p \cdot \omega, (p+1) \cdot \omega]|$ // contribution of query to v to load of box p
    if $s.max\_child\_load[p] + a_p > 1$ **then**
      return $\emptyset$
    end if
    if $t^v_{end} \geq (p+1)\omega$ **then**
      $s[p] \leftarrow s_{orig}[p] + a_p$ // add load from children in box p
      if $s[p] > 1$ **then** //insertion failed
        return $\emptyset$
      end if
    else
      $s[p] \leftarrow a_p$
    end if
  end for
  return $\{s\}$
end procedure

For each vertex $v \in V$, we show by induction on the tree size that upon termination of procedure BUILDSTRATEGY(v, ω), the returned variable $\hat{\mathcal{S}}[v]$ is the set of all *minimal* schedules $(s_v, t_v) \in \hat{\mathcal{S}}[v]$ which can be extended within the subtree T_v to a data structure $(s_u, t_u)_{u \in V(T_v)}$, for some $(s_u, t_u) \in \hat{\mathcal{S}}[u]$, $u \in V(T_v)$, in such a way that the conditions (3.5), (3.6), and (3.7) hold within subtree T_v. Here, minimality of a schedule is a trivial technical assumption, understood in the sense of the following very restrictive partial order: we say $(s_v, t_v) \leq (s'_v, t'_v)$ if $s_v[p] \leq s'_v[p]$ for all $0 \leq p \leq L-1$ and $t_v = t'_v$. (In the pseudocode, rather than write (s_v, t_v) as a pair variable, we include t_v within the structure s_v as its special field $s_v.t_v$.)

The algorithm proceeds to merge together exhaustively all possible choices of schedules $(s_{v_i}, t_{v_i}) \in \hat{\mathcal{S}}[v_i]$ of all children v_i of v, $1 \leq i \leq l$. The merge is performed by computing, for any fixed choice $(s_{v_i}, t_{v_i})_{1 \leq i \leq l}$, the combined load of each box in the resultant schedule s:

$$s[p] \leftarrow \sum_{i=1}^{l} s_{v_i}[p], \tag{3.8}$$

where, as a technicality, we also put $s[p] \leftarrow +\infty$ whenever we obtain excessive load in a box ($s[p] > 1$), as to avoid inflating the size of the state space and consequently, the running time of the algorithm. In Algorithm 5, the computation of $s[p]$ through the sum (3.8) proceeds by a processing of successive children v_i, $1 \leq i \leq l$, so that a schedule s stored in the data structure $\hat{\mathcal{S}}_i^*$ represents $s[p] = \sum_{i=1}^{i} s_{v_j}[p]$. The summation of load is performed within the subroutine MERGESCHEDULES, which merges a schedule $s_{orig} \in \hat{\mathcal{S}}_{i-1}^*$ with a schedule $s_{add} \in \hat{\mathcal{S}}[v_i]$ to obtain the new schedule $s \in \hat{\mathcal{S}}_i^*$.

Eventually, the set of schedules $\hat{\mathcal{S}}_l^*$, obtained after merging the schedules of all children of v, contains an element s satisfying (3.8). Next, we test all possible values of $t_v \in \mathbb{R} \cup \{\perp\}$, which are feasible for an aligned schedule. These values depend on whether vertex v is heavy or light, for which t_v should represent the starting time of a box or slot, respectively. Using procedure INSERTVERTEX, we then set the load of each box following (3.6):

$$s_v[p] \leftarrow \begin{cases} a_p + \sum_{j=1}^{l} s_{v_j}[p], & \text{when } t_{end}^v \geq (p+1)\omega, \\ a_p, & \text{when } p\omega < t_{end}^v < (p+1)\omega, \\ 0, & \text{when } t_{end}^v \leq p\omega, \end{cases} \tag{3.9}$$

where a_p is defined as in (3.4). In the pseudocode of function INSERTVERTEX, for compactness we replace the second and third condition by equivalently setting $s_v[p] \leftarrow a_p$ when the first condition does not hold. We additionally constrain in procedures MERGESCHEDULES and INSERTVERTEX the possibility of the condition $t_v = \perp$ occurring by enforcing the constraints of (3.5) (corresponding of the setting of parameter $s.must\_contain\_v$ to $false$). Condition (3.7) is enforced through procedures MERGESCHEDULES and INSERTVERTEX using the auxiliary array $s.max\_child\_load[p]$, $0 \leq p \leq L-1$, defined so that $s.max\_child\_load[p] \leftarrow \max_{1 \leq j \leq l} s_{v_j}[p]$.

Since $\hat{\mathcal{S}}[v_i]$, for all $1 \leq i \leq l$, contains all minimal schedules satisfying (3.5), (3.6), and (3.7), the same holds for $\hat{\mathcal{S}}[v]$, which was constructed by enforcing only

the required constraints. We remark that we obtain only the set of minimal (and not all) schedules due to the slight difference between (3.9) and (3.6) in the second condition: instead of requiring $s_v[p] \geq a_p$, we put $s_v[p] \leftarrow a_p$, thus setting the p-th coordinate of the schedule at its minimum possible value. □

It follows directly from Lemma 3.4.3 that, for any value ω^*, tree T may only admit an aligned schedule assignment of duration at most $\omega^* L$ if a call to procedure BUILDSTRATEGY$(r(T), \omega^*)$ returns a non-empty set. Taking into account Lemmas 3.4.1 and 3.4.2, we directly obtain the following lower bound on the length of the shortest aligned schedule in tree T'.

Lemma 3.4.4. *If* BUILDSTRATEGY$(r(T), \omega^*) = \emptyset$*, then:*

$$\omega^* L < \left(1 + \frac{3}{c}\right) \texttt{OPT}(T') \leq \left(1 + \frac{3}{c}\right)\left(1 + \frac{2}{c}\right) \texttt{OPT}(T) \leq \left(1 + \frac{11}{c}\right) \texttt{OPT}(T).$$

Finally, we bound the running time of procedure BUILDSTRATEGY.

Lemma 3.4.5. *The running time of procedure* BUILDSTRATEGY$(r(T), \omega)$ *is at most* $O((cn)^{\gamma L})$*, for some absolute constant* $\gamma = O(1)$*, for any* $\omega \leq n$*.*

Proof. The procedure BUILDSTRATEGY is run recursively, and is executed once for each node of the tree. The time of each execution is upper-bounded, up to multiplicative factors polynomial in n, by the size of the largest of the schedule sets named $\hat{\mathcal{S}}[u]$, $u \in V$, or $\hat{\mathcal{S}}_i^*$, appearing in the procedure. We further focus only on bounding the size $|\hat{\mathcal{S}}|$ of the state space of distinct possible schedules in the (s_v, t_v) representation. The array s_v has size L, with each entry $s_v[p]$, $0 \leq p \leq L-1$, taking one of the values $s_v[p] \in \{0, \frac{1}{\omega cn}, \frac{2}{\omega cn}, \ldots, 1\}$, where the size of the set of possible values is $\omega cn+1 \in \mathbb{N}$. Additionally, in some of the auxiliary schedules, the additional array field $s_v.max\_child\_load$ has length L, with each entry $s_v.max\_child\_load[p]$, $0 \leq p \leq L-1$, likewise taking one of the values from the set $\{0, \frac{1}{\omega cn}, \frac{2}{\omega cn}, \ldots, 1\}$. Finally, for the time t_v, we have: $t_v \in \{0, \omega, 2\omega, \ldots, (L-1)\omega, \perp\}$, where the size of the set of possible values is $L+1$.

Overall, we obtain:

$$|\hat{\mathcal{S}}| \leq (L+1)\left(\omega cn + 1\right)^L \left(\omega cn + 1\right)^L \leq (L+1)\left(cn^2 + 1\right)^{2L} < (cn)^{L\gamma'},$$

where $\gamma' > 0$ is a suitably chosen absolute constant. Accommodating the earlier omitted multiplicative $O(\text{poly}(n))$ factors in the running time of the algorithm, we get the claim for some suitably chosen absolute constant $\gamma > \gamma'$. □

3.4.4 Sequence Assignment Algorithm with Small $\texttt{COST}^{(\omega,c)}$

The procedure for computing a sequence assignment S which achieves a small value of $\texttt{COST}^{(\omega,c)}$ is given in Algorithm 7.

Algorithm 7 Construction of sequence assignment S

$\omega \leftarrow \frac{1}{cn}$
while BUILDSTRATEGY$(r(T), \omega) = \emptyset$ **do**
    $\omega \leftarrow \omega + \frac{1}{cn}$
end while
$(s_v, t_v)_{v \in V} \leftarrow$ schedule assignment of duration at most $L\omega$, satisfying constraints (3.5), (3.6), and (3.7),
    reconstructed by backtracking through the sets $(\hat{\mathcal{S}}[v])_{v \in V}$ computed in the last call to procedure BUILDSTRATEGY$(r(T), \omega)$.
for $v \in V$ **do**
    $C(v) \leftarrow \emptyset$
    for $u \in V(T_v)$ **do**
        if there is no vertex $z \neq u$ on the path from v to u s.t. $t_z < \lfloor t_u + w'(u) \rfloor_\omega + \omega$ **then**
            $C(v) \leftarrow C(v) \cup \{(\lfloor t_u \rfloor_\omega, \lceil t_u + w'(u) \rceil_\omega, u)\}$
        end if
    end for
    $S(v) \leftarrow$ sequence of vertices (third field) of $C(v)$ sorted in non-decreasing order, with tuples compared by first field, then second field, then third field.
end for
return $(S(v))_{v \in V}$

We start by observing in Algorithm 7 that if a node v is not queried ($t_v = \perp$), then all of the children of v belong to the schedules produced by procedure BUILDSTRATEGY following condition (3.5), and thus they will also appear in $S(v)$. This guarantees the validity of the solution.

Lemma 3.4.6. *Algorithm 7 returns a correct query sequence assignment S for tree T.*

For the purposes of analysis, we extend the notion of backtracking procedure BUILDSTRATEGY in a natural way, so that, for every node $v \in V$ and box $0 \leq p \leq L-1$, we describe precisely the contribution $c_v[p,u]$ of each vertex $u \in V(T_v)$ to the load $s_v[p]$. (See Fig. 3.2 for an illustration.) Formally, for $u = v$ we have $c_v[p,v] \leftarrow a_p = |[t_v, t_v + w'(v)] \cap [p\omega, (p+1)\omega]|$ if $t_v \neq \perp$, and $c_v[p,v] \leftarrow 0$, otherwise. Next, if $u \neq v$ and u belongs to the subtree of child v_i of v, we put:

$$c_v[p,u] \leftarrow \begin{cases} c_{v_i}[p,u], & \text{if } t^v_{end} > p\omega, \\ 0, & \text{otherwise,} \end{cases}$$

where the insertion time t^v_{end} for v is defined as in Observation 3.4.1. Comparing with (3.9), we have directly for all $0 \leq p < L$:

$$s_v[p] = \sum_{u \in V(T_v)} c_v[p,u].$$

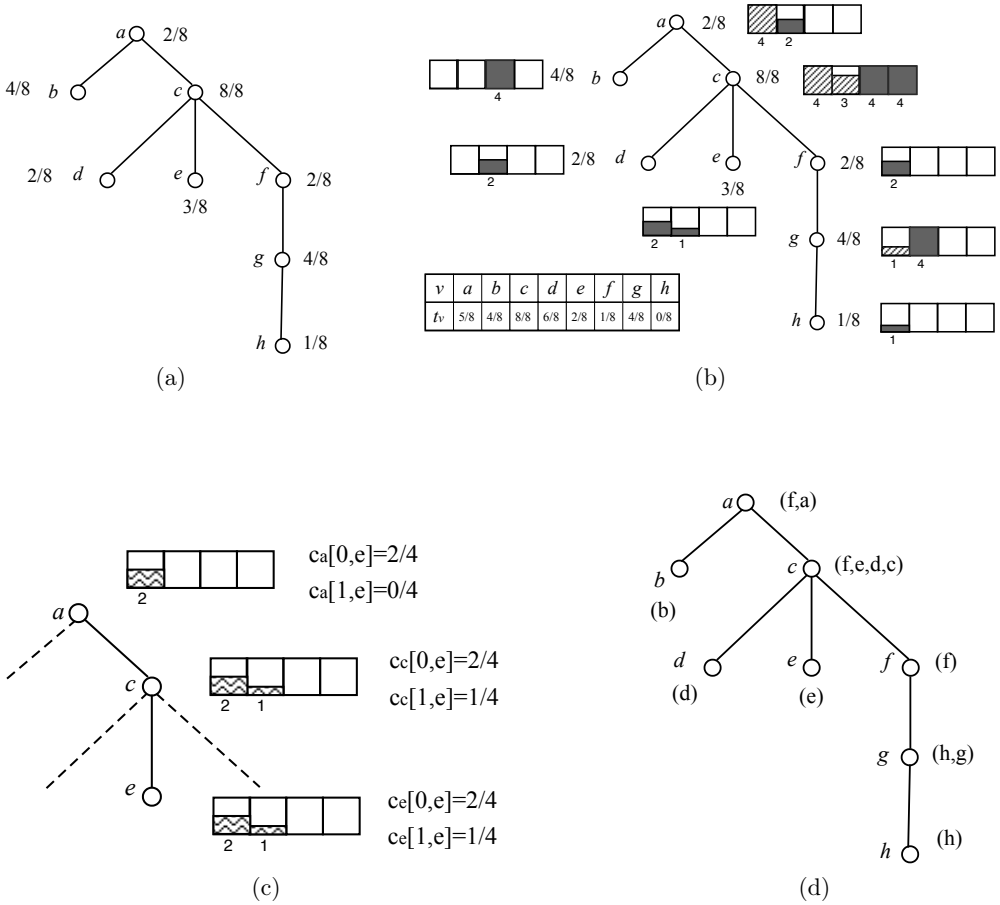


Figure 3.2: Illustration of Algorithm 2 and 3. The depicted tree T' has vertex set $V = \{a, b, c, d, e, f, g, h\}$ and vertex weights:

v	a	b	c	d	e	f	g	h
$w'(v)$	$\frac{2}{8}$	$\frac{4}{8}$	$\frac{8}{8}$	$\frac{2}{8}$	$\frac{3}{8}$	$\frac{2}{8}$	$\frac{4}{8}$	$\frac{1}{8}$

(a) Tree T' with vertex weights.
(b) Sample schedule $(s_v, t_v)_{v \in V}$ obtained by backtracking procedure BuildStrategy with parameters $c = 1, n = 8, \omega = \frac{4}{8}$, (box size $\frac{4}{8}$, slot size $\frac{1}{8}$, 4 slots per box), $L = 4$. Note that the schedules $(s_v)_{v \in V}$ may correspond to different starting times of jobs within the prescribed box; the provided t_v are an example.
(c) Contribution of load of vertex e to different vertices of the tree.
(d) Sequences $S(v)$ computed by Algorithm 3 based on provided $(s_v, t_v)_{v \in V}$. Note that vertex e does not appear in $S(a)$ because of the query to c on the way.

Let $p_s(u)$ and $p_f(u)$ be the indices of the starting and final box, respectively, to which vertex u adds load, formally $p_s(u) = \min P_u$ and $p_f(u) = \max P_u$, where $P_u = \{p : |[t_u, t_u + w'(u)] \cap [p\omega, (p+1)\omega]| > 0\}$. From the statement of Algorithm 7, we show immediately by inductive bottom-up argument that if $u \in S(v)$, then $\omega \sum_{p=p_s(u)}^{p_f(u)} c_v[p, u] = w'(u)$.

Lemma 3.4.7. *Let $(s_v, t_v)_{v \in T(V)}$ be a schedule assignment computed by* BuildStrategy. *For any vertices u and z such that (u, t_u) and (z, t_z) belong to the schedule at v, if either u or z is heavy, then $|[t_u, t_u + w'(u)] \cap [t_z, t_z + w'(z)]| = 0$.*

Proof. Note that procedure InsertVertex is called for a heavy input vertex with its last parameter (insertion time) being a multiple of ω, and the weight $w'(u)$ is a multiple of ω by definition. Thus, the interval $[t_u, t_u + w'(u)]$ starts and ends at the beginning and end of a box, respectively. Hence, Constraint (3.7) gives the lemma. □

As a consequence of Lemma 3.4.7, if these two jobs (u, t_u) and (z, t_z) overlap, where u and z belong to the sequence assignment $S(v)$, then both of the vertices u and z must be light, thus:

$$t_u > t_z - w'(u) - \omega = t_z + w'(z) - w'(z) - w'(u) - \omega \geq t_z + w'(z) - (2c+1)\omega.$$

We now define the measure of progress $M(x, i)$ of strategy $\mathcal{A}_S$ when searching for target x after i queries as follows. Let Q_i be the set of the first i queried vertices. Let v_i be the current root of the tree, $v_i = r(T\langle Q_i, x\rangle)$. Let $S_i(v) \subseteq S(v)$ be the subsequence (suffix) of $S(v)$ consisting of those vertices which have not yet been queried. Now, we define:

$$M(x, i) = \begin{cases} \min_{u \in S_i(v_i)} p_s(u), & \text{if } S_i(v_i) \neq \emptyset, \\ L, & \text{if } S_i(v_i) = \emptyset. \end{cases}$$

We have by definition, $M(x, i) \in \{0, 1, \ldots, L-1, L\}$. We obtain the next Lemma from a following straightforward analysis of the measure of progress: every time following sequence $S(v)$ we successively complete queries with an 'up' result with a total duration of at least a boxes, since the queried vertices are ordered in the first place according to minimum query time, and in the second place according to query duration, the value of the minimum $p_s(u)$, for $u \in S(v)$ remaining to be queried, advances by at least a boxes.

Lemma 3.4.8. *The measure of progress $M(x, i)$ has the following properties:*

1. *If the $(i+1)$-st query returns an 'up' result, then $M(x, i+1) \geq M(x, i)$.*
2. *If the $(i+1)$-st query returns a 'down' result, then $M(x, i+1) \geq M(x, i) - (2c+1)\omega$.*
3. *Suppose that between some two steps of the strategy, $i_2 > i_1$, each of the queries $(q_{i_1+1}, \ldots, q_{i_2})$ returns an 'up' result, and moreover, the total cost of queries performed was at least $a\omega$, for some $a \in \mathbb{N}$:*

$$\sum_{j=i_1+1}^{i_2} w'(q_j) \geq a\omega,$$

where $q_j = \mathtt{Q}_{\mathcal{A}_S,j}(T,x)$. *Then,* $M(x,i_2) \geq M(x,i_1) + a$.

□

Since the value of $M(x,i)$ is bounded from above by L, we obtain from Lemma 3.4.8 that the strategy $\mathcal{A}_S$ necessarily terminates when looking for target x with cost at most $L\omega + (2c+1)\omega d_x$,

$$\mathtt{COST}_{\mathcal{A}_S}(T',x) \leq L\omega + (2c+1)\omega d_x.$$

Thus, due to the definition of $\mathtt{COST}^{(\omega,c)}$ in (3.1) and the monotonicity of of the cost of a strategy with respect to vertex weights, we obtain the following:

Corollary 3.4.2. *For the sequence assignment computed by Algorithm 7 it holds*

$$\mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) \leq \mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T') \leq \omega L.$$

□

To prove Proposition 3.4.1, it remains to show only the stability of the sequence assignment S.

Lemma 3.4.9. *The query sequence assignment S obtained by Algorithm 7 is stable.*

Proof. We perform the proof by induction. Following the definition of stability, assume that v is the root of the remaining subtree at some moment of executing $\mathcal{A}_S$ on T', and let u be a vertex such that u is a child of v lying on the path from v to the target x. We will show that following $S(u)$ always results in a subsequence of the sequence of queries performed by following $S(v)$.

Let $S^+(v)$ be the subsequence of vertices of $S(v)$ which lie in T_u, and let $S^-(v)$ be the subsequence of all remaining vertices of $S(v)$. Note that x belongs to T_u and hence any query to a node in $S^-(v)$ gives an 'up' reply.

We now observe the first (leftmost) difference v' of the sequences $S^+(v)$ and $S(u)$. Suppose that before such a difference occurs, the common fragment of the sequences contains a query to any vertex y on the path from u to x. Then, the root of both trees moves to the same child of y, and the process continues identically regardless of the initial root of the tree. Thus, such a vertex y cannot occur prior the difference in sequences $S^+(v)$ and $S(u)$.

Next, suppose that the first difference between the two sequences consists in the appearance of vertex v in sequence $S^+(v)$, i.e., $v' = v$. Then, the root of the tree moves from v to u, and the two processes proceed identically as required. This also implies that $t_v > t_u$.

Finally, we observe that no other first difference between the sequences $S^+(v)$ and $S(u)$ is possible by the formulation of Algorithm 7. In particular, if a triple $(\lfloor t_z \rfloor_\omega, \lceil t_z + w'(z) \rceil_\omega, z)$ is added to $C(u)$ in line 10, then the condition in line 9 and $t_v > t_u$ imply that the triple $(\lfloor t_z \rfloor_\omega, \lceil t_z + w'(z) \rceil_\omega, z)$ is added also to the set $C(v)$. Similarly, an insertion of a triple $(\lfloor t_z \rfloor_\omega, \lceil t_z + w'(z) \rceil_\omega, z)$ for $z \in V(T_u)$ into $C(v)$ implies that this triple also belongs to $C(u)$. Due to the sorting performed in line 13 of Algorithm 7, $S^+(v) = S(u)$.

The eventual deterministic coupling, which is obtained in all cases for the strategies starting at v and u, extends by induction to the execution of $\mathcal{A}_S$ for trees rooted at a vertex v and its arbitrary descendant u' lying on the path from v to x, hence the claim holds. □

For the chosen value ω, we can apply Lemma 3.4.4 with $\omega^* = \omega - \frac{1}{cn}$, obtaining:

$$\left(\omega - \frac{1}{cn}\right) L = \omega^* L < \left(1 + \frac{11}{c}\right) \texttt{OPT}(T),$$

thus, by Corollary 3.4.2,

$$\texttt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) \leq \left(1 + \frac{11}{c}\right) \texttt{OPT}(T) + \frac{L}{cn} \leq \left(1 + \frac{12}{c}\right) \texttt{OPT}(T),$$

where we took into account that trivially $L \leq n$ and $\texttt{OPT}(T) \geq 1$. We thus, by Lemmas 3.4.6, and 3.4.9 obtain the claim of Proposition 3.4.1.

3.4.5 Reducing the Number of Down-Queries

We start with defining a function $\ell \colon V \to \{1, \ldots, \lceil \log_2 n \rceil\}$ which in the following will be called a *labeling of* T and the value $\ell(v)$ is called the *label of* v. We say that a subset of nodes $H \subseteq V$ is an *extended heavy part* in T if $H = \{v\} \cup H'$, where all nodes in H' are heavy, no node in H' has a heavy neighbor in T that does not belong to H' and v is the parent of some node in H'. Let $H_1, \ldots, H_l$ be all extended heavy parts in T. Obtain a tree $T_C = (V_C, E_C)$ by contracting, in T, the subgraph H_i into a node denoted by h_i for each $i \in \{1, \ldots, l\}$. In the tree T_C, we want to find its labeling $\ell' \colon V_C \to \{1, \ldots, \lceil \log_2 |V_C| \rceil\}$ that satisfies the following condition: for each two nodes u and v in V_C with $\ell'(u) = \ell'(v)$, the path between u and v has a node z satisfying $\ell'(z) < \ell'(u)$. One can obtain such a labeling by a following procedure that takes a subtree T'_C of T_C and an integer i as an input. Find a central node v in T'_C, set $\ell'(v) = i$ and call the procedure for each subtree T''_C of $T'_C - v$ with input T''_C and $i + 1$. The procedure is initially called for input T and $i = 1$. We also remark that, alternatively, such a labeling can be obtained via vertex rankings [50, 81].

Once the labeling ℓ' of T_C is constructed, we extend it to a labeling ℓ of T in such a way that for each node v of T we set $\ell(v) = \ell'(v)$ if $v \notin H_1 \cup \cdots \cup H_l$ and $\ell(v) = \ell'(h_i)$ if $v \in H_i$, $i \in \{1, \ldots, l\}$.

Having the labeling ℓ of T, we are ready to define a query sequence $R(v)$ for each node $v \in V$. The $R(v)$ contains all nodes u from T_v such that $\ell(u) < \ell(v)$ and each internal node z of the path connecting v and u in T satisfies $\ell(z) > \ell(u)$. Additionally, the nodes in $R(v)$ are ordered by increasing values of their labels. See Figure 3.3 for an example.

We start by making some simple observations regarding the sequence assignment R.

Observation 3.4.3. *For each $v \in V$ and for each $u \in R(v)$, $w(u) \leq c\omega$.*

Observation 3.4.4. *For each $v \in V$, any two nodes in $R(v)$ have different labels.*

Observation 3.4.5. *The sequence assignment R can be computed in time $O(n \log n)$.*

By x we refer to the target node in T. Fix S to be a stable sequence assignment in the remaining part of this section and by R we refer to the sequence assignment constructed above. Then, we fix S^+ to be $S^+(v) = R(v) \circ S(v)$ for each $v \in V$. Denote by U_i the first i nodes queried by $\mathcal{A}_{S^+}$ and let $C_i = \min \ell(T\langle U_{i-1}, x\rangle)$ for each $i \geq 1$. For brevity we denote $U_0 = \emptyset$ and $C_0 = 0$; we also denote by u_i the node in $U_i \setminus U_{i-1}$, $i \geq 1$. A query made by $\mathcal{A}_{S^+}$ to a node that belongs to $R(v)$ for some $v \in V$ is called an *R-query*; otherwise it is an *S-query*.

Lemma 3.4.10. *For each $i \geq 0$, the nodes in $T\langle U_i, x\rangle$ with minimum label induce a connected subtree.*

The next two lemmas will be used to conclude that the number of light queries performed by $\mathcal{A}_{S^+}$ is bounded by $2 \log_2 n$ (see Lemma 3.4.13).

Lemma 3.4.11. *If the i-th query of $\mathcal{A}_{S^+}$ is an R-query resulting in an 'up' reply, then $C_{i+1} \geq C_i + 1$.*

Proof. By construction, u_i has the minimum label among all nodes in $T\langle U_{i-1}, x\rangle$. By Lemma 3.4.10, either u_i is the unique node with label $\ell(u_i)$ in the tree $T\langle U_{i-1}, x\rangle$ or there are more nodes with this label and they all belong to a single extended heavy part in $T\langle U_{i-1}, x\rangle$ with u_i being closest to the root of $T\langle U_{i-1}, x\rangle$. In both cases, since the reply is 'up', we obtain that $T\langle U_i, x\rangle$ has no node with label $\ell(u)$, which proves the lemma. □

Lemma 3.4.12. *If the i-th query of $\mathcal{A}_{S^+}$ is an R-query that results in a 'down' reply, then one of the two cases holds:*

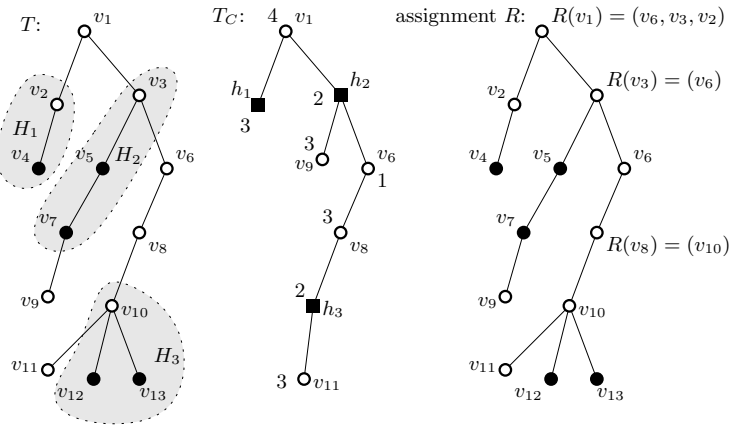


Figure 3.3: A tree T (on the left) has light vertices (marked as white nodes) and heavy ones (dark circles); also heavy extended parts are marked. The tree T_C (in the middle) is used together with its labeling (integers are the labels) to obtain the sequence assignment R (on the right); here we skip the sequence assignment for each node v for which $R(v) = \emptyset$.

(i) if u_i has no heavy child that belongs to $T\langle U_i, x\rangle$, then $C_{i+1} \geq C_i + 1$,

(ii) if u_i has a heavy child that belongs to $T\langle U_i, x\rangle$, then $C_{i+1} = C_i = \ell(u_i)$ and for each $j > i$ such that $C_i = C_{j+1}$, all queries $i+1, \ldots, j$ are S-queries.

Proof. By Lemma 3.4.10, the nodes with label C_i induce a connected subtree in $T\langle U_{i-1}, x\rangle$. This immediately implies i. By construction, if u_i has a heavy child u' that is in $T\langle U_i, x\rangle$, then $u' = r(T\langle U_i, x\rangle)$ and the labels of u_i and u' are the same. The latter is due to the fact that both u_i and u' belong to the same extended heavy part in T. Suppose for a contradiction that the j-th query (performed say on a node z) is an R-query and $C_{j+1} = C_i$, $j > i$. This in particular implies that $z \in R(r(T\langle U_{j-1}, x\rangle))$. Due to Lemma 3.4.11, the reply to this query is 'down'. By i, z has a heavy child that belongs to $T\langle U_j, x\rangle$. By Observation 3.4.3, z is a light node and therefore z along with some of its descendants and u_i with some of its descendants form two different extended heavy parts in T. Since z and u_i have the same label, there exists a light node u_i' in T on the path between u_i and z with label smaller than $\ell(u_i)$. Assume without loss of generality that no other node of this path that lies between u_i and u_i' has label smaller than $\ell(u_i')$. The above-mentioned path is contained in $T\langle U_{i-1}, x\rangle$ since both u_i and z belong to this subtree. This however implies that $u_i' \in R(v)$ because $\ell(u_i') < \ell(u_i)$ and no node on the path between u_i and u_i') has label smaller than $\ell(u_i')$. Moreover, u_i' precedes u_i in $R(v)$ meaning that among one for the first i queries, u_i' must have been queried — a contradiction with the fact that u_i' belongs to $T\langle U_i, x\rangle$. □

Lemma 3.4.13. *For each target node, the total number of R-queries made by $\mathcal{A}_{S^+}$ is at most $2\log_2 n$.*

Proof. It follows from Lemmas 3.4.11 and 3.4.12 that after any two subsequent R-queries the value of parameter C_i increases by at least 1. □

The next two lemmas will be used to bound the number of S-queries in S^+ receiving a 'down' reply to be at most $2\log_2 n$.

Lemma 3.4.14. *If all nodes in $R(v)$ have been queried by $\mathcal{A}_{S^+}$ after an i-th query for some $v \in V$ and v is the root of $T\langle U_i, x\rangle$, then $\ell(v) = C_{i+1}$.*

Proof. Suppose for a contradiction that $\ell(v) \neq C_{i+1}$. Since v belongs to $T\langle U_i, x\rangle$, we have that $\ell(v) > C_{i+1}$. Thus, by construction, there exists a light node u in $T\langle U_i, x\rangle$ with $\ell(u) = C_{i+1}$ such that all internal nodes on the path between v and u have labels larger than $\ell(u)$. Therefore, u belongs to $R(v)$ because v is the root of $T\langle U_i, x\rangle$. This implies that u has been already queried — a contradiction with u being in $T\langle U_i, x\rangle$. □

Lemma 3.4.15. *If the i-th query of $\mathcal{A}_{S^+}$ is an S-query performed on a light node and the reply is 'down', then $T\langle U_i, x\rangle$ has no light node with label C_i.*

Proof. Suppose that the i-th query is performed on a node u in $S(v)$ for some $v \in V$. Clearly, v is the root of $T\langle U_{i-1}, x\rangle$. Since the considered query is an S-query, all vertices in $R(v)$ have been already queried. Thus, by Lemma 3.4.14, $\ell(v) = C_i$. By construction, v is the only light node in this subtree having label C_i. Since the reply to the i-th query is 'down', v does not belong to $T\langle U_i, x\rangle$. □

We are now ready to prove Proposition 3.4.2.

By Lemma 3.4.13, in $\mathcal{A}_{S^+}$, the total number of R-queries does not exceed $2\log_2 n$. Note that since S is stable, for each target node x, the S-queries performed by $\mathcal{A}_{S^+}$ are a subsequence of the queries performed by $\mathcal{A}_S$. Therefore, the potentially additional queries made by $\mathcal{A}_{S^+}$ with respect to $\mathcal{A}_S$ are R-queries. By Observation 3.4.3, each R-query is made on a light node. By definition of function $\texttt{COST}^{(\omega,c)}$ and Observation 3.4.3, any R-query increases the value of $\texttt{COST}^{(\omega,c)}$ of $\mathcal{A}_{S^+}$ with respect to the value of $\texttt{COST}^{(\omega,c)}$ of $\mathcal{A}_S$ by at most $(2c+1)\omega$. Hence we have:

$$\texttt{COST}^{(\omega,c)}_{\mathcal{A}_{S^+}}(T) \leq \texttt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) + 2(2c+1)\omega\log_2 n.$$

By Lemmas 3.4.12 and 3.4.15,

the total number of queries in strategy $\mathcal{A}_{S^+}$ to light nodes receiving 'down' replies can be likewise bounded by $2\log_2 n$. Since each such query introduces a rounding difference of at most $(2c+1)\omega$ when comparing cost functions $\texttt{COST}$ and $\texttt{COST}^{(\omega,c)}$, we thus obtain:

$$\texttt{COST}_{\mathcal{A}_{S^+}}(T) \leq \texttt{COST}^{(\omega,c)}_{\mathcal{A}_{S^+}}(T) + 2(2c+1)\omega\log_2 n.$$

Combining the above observations gives the claim of the Proposition.

3.5 $O(\sqrt{\log n})$-approximation algorithm

We now present the second main result of this work. By recursively applying the previously designed QPTAS (Theorem 3.4.1), we obtain a polynomial-time $O(\sqrt{\log n})$-approximation algorithm for finding search strategy for an arbitrary weighted tree. We start by informally sketching the algorithm — we follow here the general outline of the idea from [24]. The algorithm is recursive and starts by finding a minimal subtree T^* of an input tree whose removal disconnects T into subtrees, each of size bounded by $n/2^{\sqrt{\log n}}$. The tree T^* will be processed by our optimal algorithm described in Section 3.4. This results either in locating the target node, if it belongs to T^*, or identifying the component of $T - T^*$ containing the target, in which case the search continues recursively in the component. However, for the final algorithm to have polynomial running time, the tree T^* needs to be of size $2^{O(\sqrt{\log n})}$. This is obtained by contracting paths in T^* (each vertex of the path has at most two neighbors in T^*) into single nodes having appropriately chosen weights. Since T^* has $2^{O(\sqrt{\log n})}$ leaves, this narrows down the size of T^* to the required level and we argue that an optimal search strategy for the 'contracted' T^* provides a search strategy for the original T^* that is within a constant factor from the cost of T^*.

A formal exposition and analysis of the obtained algorithm is provided in Section 3.5.2.

Theorem 3.5.1. *There is a $O(\sqrt{\log n})$-approximation polynomial time algorithm for the weighted tree search problem.*

3.5.1 Partition of a Tree

We start with some notation. Given a tree $T = (V, E, w)$ and a fixed value of parameter α, we find a subtree $T^* = (V^*, E^*)$ of the input tree T, called an *α-separating tree*, that satisfies: $r(T^*) = r(T)$ and each connected component of $T \setminus V^*$ has at most α vertices. An α-separating tree T^* is *minimal* if the removal of any leaf from T^* gives an induced tree that is not an α-separating tree. Then, for a target node $x \in V$, we introduce a recursive strategy $\mathcal{R}$ that takes the following steps:

1. $\mathcal{R}$ first applies strategy $\mathcal{A}^*$ restricted to tree T^* to locate the node x' of T^* which is closest to the target x.

2. Then, $\mathcal{R}$ queries x', which either completes the search in case when x' is the target or provides a neighbor x'' of x' that is closer to the target than x'.

3. If x' is not the target, then the strategy calls itself recursively on the subtree $T_{x''}$ of $T \setminus \{x'\}$ containing x. The latter strategy for $T_{x''}$ is denoted by $\mathcal{R}_{x''}$. (Note that $T_{x''}$ is a connected component in $T \setminus V^*$.)

Such a search strategy $\mathcal{R}$ obtained from $\mathcal{A}^*$ and strategies $\mathcal{R}_{r(T')}$ (constructed recursively) for subtrees T' in $T \setminus V^*$ is called a $(\mathcal{A}^*, \{\mathcal{R}_{r(T')} \mid T' \in \mathcal{C}(T \setminus V^*)\})$*-strategy*, where $\mathcal{C}(T \setminus V^*)$ is the set of connected components (subtrees) in $T \setminus V^*$.

The following bound on the cost of the strategy $\mathcal{R}$ follows directly from the construction:

Lemma 3.5.1. *For a $(\mathcal{A}^*, \{\mathcal{R}_{r(T')} \mid T' \in \mathcal{C}(T \setminus V^*)\})$-strategy $\mathcal{R}$ for T it holds*

$$\mathtt{COST}_{\mathcal{R}}(T) \leq \mathtt{COST}_{\mathcal{A}^*}(T^*) + \max_{x' \in V^*} w(x') + \max_{T' \in \mathcal{C}(T \setminus V^*)} \mathtt{COST}_{\mathcal{R}_{r(T')}}(T').$$

□

We now formally describe and analyze the aforementioned contractions of subpaths in a tree. A maximal path with more than one node in a tree T that consists only of vertices that have degree two in T is called a *long chain in T*. For each long chain P, contract it into a single node v_P with weight $\min_{u \in V(P)} w(u)$, obtaining a tree $\xi(T)$. In what follows, the tree $\xi(T)$ is called a *chain-contraction of T*.

Our first step is a remark that, at the cost of losing a multiplicative constant in the final approximation ratio, we may restrict ourselves to trees that have no long chains. This is due to the following observation.

Lemma 3.5.2. *Let T be a tree. Given a p-approximate search strategy for $\xi(T)$, a $(p+1)$-approximate search strategy for T can be computed in polynomial time.*

Proof. Let $\mathcal{A}'$ be a search strategy for $\xi(T)$. We obtain a search strategy $\mathcal{A}$ for T in two stages. In the first stage we 'mimic' the behavior of $\mathcal{A}'$: (i) if $\mathcal{A}'$ queries a node v that also belongs to T, then $\mathcal{A}$ also queries v; (ii) if $\mathcal{A}'$ queries a node v_P that corresponds to some long chain P in T, then $\mathcal{A}$ queries, in T, a node with minimum weight in P. Note that after the first stage, the search strategy either located the target or determined that the target belongs to a subpath P' of some long chain P

of T. Moreover, the total cost of all queries performed in the first stage is at most $\texttt{COST}_{\mathcal{A}'}(\xi(T))$.

Then, in the second stage we compute (in $O(n^2)$-time) an optimal search strategy $\mathcal{A}_{P'}$ for P' [23]. Due to the monotonicity of the cost over taking subgraphs, $\texttt{COST}_{\mathcal{A}_{P'}}(P') = \texttt{OPT}(P') \leq \texttt{OPT}(T)$.

Both stages provide us with a search strategy for T with cost at most $\texttt{COST}_{\mathcal{A}'}(\xi(T)) + \texttt{OPT}(T)$. Since, $\texttt{OPT}(\xi(T)) \leq \texttt{OPT}(T)$ and $\texttt{COST}_{\mathcal{A}'}(\xi(T)) \leq p \cdot \texttt{OPT}(\xi(T))$, the lemma follows. □

Note that it is straightforward to verify whether any vertex v of T is a leaf in the α-separating tree of T and hence we obtain the following.

Observation 3.5.1. *Given a tree T with no long chain and α, a minimal α-separating tree of T can be computed in polynomial-time.* □

Using Lemma 3.5.2 and choosing appropriately the value of α, one can obtain an α-separating tree of T having at most $t = 2^{O(\sqrt{\log n})}$ vertices.

Lemma 3.5.3. *Let T be any tree and let α be selected arbitrarily. If T^* is a minimal α-separating tree of T, then $\xi(T^*)$ has at most $4\left\lceil \frac{n}{\alpha} \right\rceil$ vertices.*

Proof. By definition, for each leaf v of T^*, the subtree T_v has more than α nodes. Since these trees are node-disjoint, we obtain that there are at most $\lceil \frac{n}{\alpha} \rceil$ leaves in T^*. We denote the leaves of T^* by $v_1, v_2, \ldots, v_l$, $l \leq \lceil \frac{n}{\alpha} \rceil$; note that $\xi(T^*)$ has the same leaves as T^*.

Let $V(\xi(T^*))$ be the vertex set of $\xi(T^*)$. Then, we claim that $|V(\xi(T^*))| = O(\lceil \frac{n}{\alpha} \rceil)$ by counting the number of nodes with different degrees in $\xi(T^*)$. Clearly, we have $\left|\{v \in V(\xi(T^*)) \;\middle|\; \deg(v) > 2\}\right| \leq \lceil \frac{n}{\alpha} \rceil$. Since the tree $\xi(T^*)$ contains no long chains, the parent (if exists) of every node with degree exactly 2 must have degree at least 3. Thus,

$$\left|\{v \in V(\xi(T^*)) \;\middle|\; \deg(v) = 2\}\right| \leq \left|\{v \in V(\xi(T^*)) \;\middle|\; \deg(v) > 2\}\right| + 1 \leq \left\lceil \frac{n}{\alpha} \right\rceil + 1.$$

Hence we get $|V(\xi(T^*))| \leq 4\left\lceil \frac{n}{\alpha} \right\rceil$. □

3.5.2 Recursive Execution of Strategy

With Lemmas 3.5.2, 3.5.3 and Observation 3.5.1 we are now ready to obtain the efficient recursive decomposition of the problem:

Lemma 3.5.4. *If there is a $O(1)$-approximation algorithm running in $n^{O(\log n)}$ time for any input tree, then one can obtain a $O(\sqrt{\log n})$-approximation algorithm with polynomial running time for any input tree.*

Proof. Suppose SOLVE is a given constant-factor approximation algorithm running in time $n^{O(\log n)}$ that, for any input tree T, outputs a search strategy for T. We then design a polynomial-time procedure REC as shown in Algorithm 8, which outputs a search strategy $\mathcal{R}$ for an input tree T.

Algorithm 8 $O(\sqrt{\log n})$-approximation procedure REC based on $n^{O(\log n)}$-time constant approximation algorithm SOLVE

```
1: procedure REC(tree T = (V, E, w))
2:     n ← |V|
3:     if n ≤ 2^√(log n) then
4:         return SOLVE(T)
5:     else
6:         α ← n/2^√(log n)
7:         T* ← a minimal α-separating tree of T with vertex set V*
8:         A* ← SOLVE(ξ(T*))
9:         A_T* ← search strategy for T* obtained from A* as described in proof of
   Lemma 3.5.2
10:        for each T' in C(T \ V*) do
11:            R_r(T') ← REC(T');
12:        end for
13:        return (A_T*, {R_r(T') | T' ∈ C(T \ V*)})-strategy for T
14:    end if
15: end procedure
```

Each call to SOLVE in line 4 has running time $(2^{\sqrt{\log n}})^{O(\log(2^{\sqrt{\log n}}))}$, which is a polynomial in n. The same holds for each call call in line 8 because, by Lemma 3.5.3, $\xi(T^*)$ has at most $4\left\lceil \frac{n}{\alpha} \right\rceil = O(2^{\sqrt{\log n}})$ vertices. Thus, procedure REC has polynomial running time and it remains to bound the cost of the search strategy $\mathcal{R}$ computed by REC.

To bound the recursion depth of REC, note that each time a recursive call is made, the size of instance (input tree) decreases $2^{\sqrt{\log n}}$ times. Thus, the depth is bounded by $\log_{(2^{\sqrt{\log n}})} n = \sqrt{\log n}$. In the search strategy computed by procedure REC, at each level of the recursion we execute the search strategy computed by one call to SOLVE and one vertex of the $(n/2^{\sqrt{n}})$-separating tree is queried. This follows from the definition of $(\mathcal{A}_{T^*}, \{\mathcal{R}_{r(T')} \mid T' \in \mathcal{C}(T \setminus V^*)\})$-strategy. By Lemma 3.5.2,

$$\texttt{COST}_{\mathcal{A}_{T^*}}(T^*) \leq c' \cdot \texttt{OPT}(T^*)$$

for some constant c'. By Lemma 3.5.1 and since $\texttt{OPT}(T^*) \leq \texttt{OPT}(T)$, the cost of $\mathcal{R}$ at each recursion level is bounded by $(c'+1)\texttt{OPT}(T)$. This gives that $\texttt{COST}_{\mathcal{R}}(T) \leq c'\sqrt{\log n} \cdot \texttt{OPT}(T)$ as required. □

Noting that the existence of a constant-approximation procedure with $n^{O(\log n)}$ running time follows from Theorem 3.4.1 (by taking $\varepsilon = 1$), the claim of Theorem 3.5.1 follows directly from Lemma 3.5.4.

3.6 Conclusions of Chapter

We consider the generalization of the binary search problem. A search strategy is required to locate an unknown target node t in a given tree T. Upon querying a

node v of the tree, the strategy receives as a reply an indication of the connected component of $T \setminus \{v\}$ containing the target t. The cost of querying each node is given by a known non-negative weight function, and the considered objective is to minimize the total query cost for a worst-case choice of the target. Designing an optimal strategy for a weighted tree search instance is known to be strongly NP-hard.

And we have the following results on weighted tree search problem:

(1) A quasi-polynomial time approximation scheme: for any $0 < \varepsilon < 1$, there exists a $(1+\varepsilon)$-approximation strategy with a computation time of $n^{O(\log n/\varepsilon^2)}$. Thus, the problem is not APX-hard, unless $NP \subseteq DTIME(n^{O(\log n)})$.

(2) By applying a generic reduction, we obtain as a corollary that the studied problem admits a polynomial-time $O(\sqrt{\log n})$-approximation. This improves previous $\hat{O}(\log n)$-approximation approaches, where the $\hat{O}$-notation disregards $O(\text{poly}\log\log n)$-factors.

Results of this work have been presented in:

- Dariusz Dereniowski, Adrian Kosowski, Przemyslaw Uznanski, Mengchuan Zou *Approximation Strategies for Generalized Binary Search in Weighted Trees.* International Colloquium on Automata, Languages, and Programming (ICALP), 2017

Chapter 4

Pure-LOCAL Weight Update Algorithm Approximating Max-flow

In this chapter we study the design of Pure-LOCAL algorithm, we implement a multiplicative weights update algorithm [20] by Mądry et al that computes a $(1+\varepsilon)$-approximation for Max-flow problem in Pure-LOCAL model. We show that our algorithm solves a weaker version of Max-flow $(1+\varepsilon)$-approximation decision problem in polynomial time (Theorem 4.4.1).

4.1 Introduction: The Maximum s-t Flow Problem

Let $G = (V, E)$ be an undirected graph, with n vertices and m edges, among which there are two special vertices, a source s and a sink t. Every edge is assigned with a positive integral capacity $U_e \in \mathbb{Z}^+$. Let A be the adjacency matrix of G, for $u \in V$ we denote $N(u) = \{v \in V | uv \in E\}$, and for $e \in E$, we denote $e \sim u$ if e is incident to u, i.e. $e = uv \in E$, for some $v \in V$.

Definition 4.1.1. *An s-t flow is a function $f : E \to \mathbb{R}$ obeying the flow-conservation constraints*

$$\sum_{e \sim u} f(e) = 0, \text{ for all } u \in V \setminus \{s, t\}$$

The value of flow $|f|$ is defined by $|f| := \sum_{e \sim s} f(e)$ and is naturally equal to $\sum_{e \sim t} f(e)$ by flow-conservation.

Here U_e is the edge capacity and we assume that is polynomial w.r.t. n. We denote the **congestion** of an edge by $cong_e(\boldsymbol{f}) = \frac{|f_e|}{U_e}$, an s-t flow is feasible for capacities U_e if $\forall e \in E$, $|f(e)| \leq U_e$, or $cong_e(\boldsymbol{f}) \leq 1$. The maximum s-t flow problem(or max-flow problem for short) is to find a feasible s-t flow with maximum value. Without loss of generality we assume that the source and sink are always the node 1 and n in our work.

Recently, a series of studies are raised on using continuous optimization method to compute $(1 + \varepsilon)$ approximation of maximum $s - t$ flow [20, 61, 68] their basic framework is to regard the maximum $s - t$ flow problem as an optimization problem

defined by a Laplacian system (which is equivalent to the description of the electrical flow in circuits) and optimize with a Laplacian solver [55], where two main optimization techniques are adopted to approximate the optimal solution: the multiplicative weights update method [4, 20] and gradient descent based method [61]. Our work is based on one of these works [20] based on multiplicative weights update method and electrical flow and we will present basics in next section.

A small but in need reminder of the notation is that we assume all vectors in this chapter are row vectors.

4.2 Preliminaries

4.2.1 Electrical Flow and Graph Laplacian

Definition 4.2.1. *We associate graph $G = (V, E)$ with resistances, namely a vector $\mathbf{r} \in \mathbb{R}^m$, by assigning $r_e > 0$ to each $e \in E$. An electrical flow is an s-t flow that accepts a vector of potentials $\phi \in \mathbb{R}^n$, such that, for $e = (u, v)$*

$$f(e) = \frac{\phi(u) - \phi(v)}{r_e}$$

For a given s-t flow f, the *energy* of f is defined by

$$\mathcal{E}(f) := \sum_{e \in E} r(e) f^2(e)$$

An electrical flow of value F is also the flow that minimizes $\mathcal{E}(f)$ among all s-t flows of value F.

We call the vector $C \in \mathbb{R}^m$ s.t. $c_{uv} = \frac{1}{r_e}$, $e = uv$ the *conductance* vector of G. Denote $A \in \mathbb{R}^{n \times n}$ to be the weighted adjacency matrix of G, s.t. $A_{uv} = c_{uv}$ if $uv \in E$, and 0 otherwise. Let $D \in \mathbb{R}^{n \times n}$, $d_{uu} = \sum_{uv \in E} c_{uv}$ and 0 for $d_{uv}, u \neq v$. Then $P = D^{-1}A$ is the transition matrix, e.g. $P_{uv} = \frac{c_{uv}}{\sum_{v \in N(u)} c_{uv}}$.

Then we introduce the definition of the Laplacian matrix.

Definition 4.2.2. *Given an simple (no loop, no multiple edges) undirected weighted graph $G = (V, E, \boldsymbol{c})$ where c_{uv} is the weight for $uv \in E$, let A be the weighted adjacency matrix, i.e. $A_{uv} = c_{uv}$ if $uv \in E$ and 0 otherwise. The **volume** of a vertex u is $vol(u) = \sum_{v \in N(u)} c_{uv}$. And let D be a diagonal matrix $D_{uu} = vol(u), u \in V$. Then the **Laplacian matrix** $L \in \mathbb{R}^{n \times n}$ of G is defined by $L = D - A$ or explicitly:*

$$L_{uv} = \begin{cases} -c_{uv} & uv \in E \\ \sum_{v \in N(u)} c_{uv} & u = v \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

A well-known correspondence between electrical flow and graph laplacian is that, for $\boldsymbol{b} \in \mathbb{R}^n$ and a constant $F \in \mathbb{R}$, such that $b_s = F$, $b_t = -F$ and $b_u = 0, u \in V \setminus \{s, t\}$, then the solution $\boldsymbol{h}$ of the Laplacian system

$$L\boldsymbol{h}^\top = \boldsymbol{b}^\top \tag{4.2}$$

is a potential vector of the electrical flow of value F induced by resistances $\boldsymbol{r}$ (or conductances C). This is from the definitions and basic circuit law and a computations is provided in [20].

That means, given an electrical network with constant current source of value F, let $\boldsymbol{r}$ be the resistances and C the conductances associated to $\boldsymbol{r}$, to compute a potential vector of this electrical network, we compute: $A \in \mathbb{R}^{n\times n}$, $A_{uv} = c_{uv}$ if $uv \in E$, and 0 otherwise. $D \in \mathbb{R}^{n\times n}$, $d_{uu} = \sum_{uv\in E} c_{uv}$ and 0 for $d_{uv}, u \neq v$, and $L = D - A$, then if we could solve the Laplacian system $L\boldsymbol{h}^\top = \boldsymbol{b}^\top$ then we can get a potential vector of the electrical network. By Definition 4.2.1 we can also compute the electrical flow of this network.

We haven seen the relationship of solving an electrical network and solving the associated laplacian system. Then we explain the use of solving electrical networks in the recent studies of solving max-flow problem by continuous optimization.

4.2.2 Electrical Flow and Max-flow Problem

We consider the decision problem of maximum $s-t$ flow, i.e. given $G = (V, E)$ and edge capacities U_e, $e \in E$, and a value F_{test}, we'd like to know if there is a feasible s-t flow $\boldsymbol{f}$, such that $|\boldsymbol{f}| \leq F_{test}$. If we could solve the decision problem, we could compute the value of the max-flow by a binary search on value F_{test}.

This decision problem is related to the following linear system:

$$\begin{aligned}
&\text{minimize } \max_e(cong_e(\boldsymbol{f})) \\
&\sum_{v\in N(u), e=uv} f(e) = 0, && \text{for all } u \in V \setminus \{s,t\} \\
&\sum_{v\in N(u), e=uv} f(e) = F && u = source \\
&f(e) \geq 0, && \forall e \in E
\end{aligned} \tag{4.3}$$

Then F_{test} could be a feasible value of a s-t flow if and only if the solution of the linear program is less than 1. Thus the decision problem of maximum $s-t$ flow, is equivalent to this special case of the minimization problem on the maximum congestion.

However, solving the linear program may not be easier than the original program, but optimization techniques could be applied here to get faster $(1+\varepsilon)$-approximation algorithms. In [20] authors provide an $(1+\varepsilon)$-approximation algorithm in nearly-linear time (linear time with polylog factors) by solving a sequence of electrical flow problems. Our work is based on this algorithm and we briefly introduce their idea here.

Remember that an electrical flow of value F is the flow that minimizes $\mathcal{E}(f)$ among all s-t flows of value F, in other words, let $\tilde{\boldsymbol{f}}$ denote the electrical flow of the given G, $\boldsymbol{r}$ and F, then $\tilde{\boldsymbol{f}}$ is exactly the optimal solution of the following quadratic program:

$$
\begin{aligned}
\text{minimize } & \sum_{e}(f^2(e)\cdot r_e) \\
& \sum_{v\in N(u),e=uv} f(e)=0, && \text{for all } u\in V\setminus\{s,t\} \\
& \sum_{v\in N(u),e=uv} f(e)=F && u=source \\
& f(e)\geq 0, && \forall e\in E
\end{aligned}
\tag{4.4}
$$

If we set r_e be proportional to $\frac{1}{U_e^2}$, then the objective function of Program 4.4 is proportional to $\sum_e(\frac{f(e)}{U_e^2}) = \sum_e cong_e^2(\boldsymbol{f})$, if we could choose r_e such that the solution of Program 4.4 close to the minimization of maximum congestion, then it can approximately solve the decision problem. Notice that in Section 4.2.2 we know that electrical network could be solved by solving a Laplacian system, so the solution of Program 4.4 could be also computed by solving the Laplacian system.

In [20], authors proposed an algorithm iteratively choosing r_e to make the solution of electrical network approaching the minimization of maximum congestion by a framework of Multiplicative Weights Update Method [4]. In fact they proved that the following procedure of adaptively choosing $\boldsymbol{r}$:

$$
\begin{aligned}
w_e^{(0)} &= 1, \forall e\in E \\
r_e^{(t)} &= \frac{1}{U_e^2}\left((w_e^{(t)}+\frac{\varepsilon\left\|\boldsymbol{w}^{(t)}\right\|_1}{3m}\right) \\
f_e^{(t)} &= \text{electrical flow induced by } \boldsymbol{r}^{(t)} \text{ and } F \\
cong(f_e^{(t)}) &= \frac{f_e^{(t)}}{U_e} \\
w_e^{(t)} &= \left(w_e^{(t-1)}\right)\left(1+\frac{\varepsilon}{\rho}cong_e(f^{(t-1)})\right)
\end{aligned}
\tag{4.5}
$$

will have a good guarantee on the maximum congestion, such that the average congestion of a sequence of electrical flow computed is bounded by $1+O(\varepsilon)$. Here ρ is a parameter of the upper bound of congestions over the iterations of the algorithm, which influences the number of iterations needed, and is going to be set later. In fact, [20] shows that here we don't need to compute the electrical flow exactly, and could be an approximate solution of the electrical network (Laplacian system). Our work has implemented this framework to Pure-LOCAL model and we provide a proof inspired in its general outline by the framework of [20] in Section 4.4.2.

Challenges of implementing in Pure-LOCAL model. To implement the Algorithm 4.5 in Pure-LOCAL model, there are two main challenges:

1. How to compute $\frac{\left\|\boldsymbol{w}^{(t)}\right\|_1}{m}$? It's not trivial because $\left\|\boldsymbol{w}^{(t)}\right\|_1$ is a global quantity and is changing between iterations.
2. How to compute the electrical flow induced by $\boldsymbol{r}^{(t)}$ and F ? There is no known general Laplacian solver in Pure-LOCAL model.

Our work provides the approach to approximate the above two quantities in Pure-LOCAL model. The idea of the approach makes use of the application of properties of random walks on graphs. We present the basics in the next section.

4.2.3 Connection to Random Walk

In the previous sections we presented the connections between the Max-flow problem, electrical flows and the Laplacian system, and [20] showed that Max-flow could be approximated by solving the electrical network (equivalent to solve the associated Laplacian system). Finally, the question remains of how to solve the electrical flow Laplacian system in Pure-LOCAL.

In centralized scenarios, works based on electrical flows [20, 61, 68] have nearly all used an Laplacian solver by Koutis et al [55], which is developed from the well-known work of Spielman and Teng [84]. However, this method is quite sophisticated and hard to implement in distributed settings. In our work, we use the matrix powers from the idea of random walks to ensure local computations.

Random Walk

Given a graph $G = (V, E)$, a random walk on G is a discrete time Markov chain $X^{(1)}, X^{(2)}, ..., X^{(k)}, ...$ on the set of vertices V with transition matrix P that defines the probability from a node to arrive another node in one step, i.e. a move generated from a start vertex by selecting an edge, traversing the edge to a new vertex, and repeating the process, such that at vertex x, the probability of next vertex being $y \in V$ is $P(x, y)$.

Mixing time

Given a discrete time Markov chain $X^{(1)}, X^{(2)}, ..., X^{(k)}, ...$ on a state space $\mathcal{X}$ with **transition matrix** P, i.e. a move such that for a given state $x \in \mathcal{X}$, the probability of next state being $y \in \mathcal{X}$ is decided by $P(x, y)$, let $\pi^{(k)}$ a row vector denote the distribution of the random variable $X^{(k)}$, we call the distribution π on $\mathcal{X}$ a **stationary distribution** [62] if

$$\pi = \pi P \tag{4.6}$$

The uniqueness and convergence to a stationary distribution for Markov chains are well-studied.

A Markov chain is **irreducible** if for any $x, y \in \mathcal{X}$, the probability of arriving at y from x at some step is positive. It is known that an irreducible Markov chain has a unique stationary distribution π [62].

For $T(x) = \{t \geq 1 : P^{(t)}(x, x) > 0\}$ the set of times when the probability of a chain starting at x go back to x is positive, the period of state x is $gcd(T(x))$. A Markov chain is **aperiodic** if for every x, the period of x is 1.

It is well-known that Convergence Theorem (Theorem 4.9 of [62]) shows that a finite irreducible aperiodic Markov chain converge to the unique stationary distribution, and we could define mixing time for the Markov chain.

Definition 4.2.3. *The mixing time of the Markov chain of transition matrix P with stationary distribution π started from state x is*

$$mix(P, x) = \min\{k : ||\pi^{(k')} - \pi||_\infty \leq 1/3, \forall k' \geq k\} \tag{4.7}$$

The mixing time of the Markov chain P with stationary distribution π is

$$mix(P) = \max_{x \in \mathcal{X}}\{mix(P, x)\} \tag{4.8}$$

Then the mixing time of a Markov chain depends only on its transition matrix P and does not depend on the initial distribution.

Scaling of the error parameter. In fact in Definition 4.2.3 the constant $1/3$ could be any constant less than $1/2$, we could also define the ε-mixing time $mix_\varepsilon(P)$ by replacing the constant $1/3$ by ε, scaling the error parameter will bring an logarithmic factor to the mixing time of a Markov chain. From Section 4.5 of [62], we have the following proposition:

Proposition 4.2.1. *For any $0 < \varepsilon < 1/2$, we have*

$$mix_\varepsilon(P) = \lceil \log_2 \varepsilon^{-1} \rceil \cdot mix(P) \tag{4.9}$$

In our further analysis of our algorithm, we will regard the mixing time as a parameter of a given transition matrix of a graph.

Our work brings the results from studies on two random walks, namely the simple random walk on graphs [67, 62] and the weighted random walk with edge conductances [62].

Simple random walk on graphs

Definition 4.2.4. *A simple random walk is a random walk on G with transition matrix P defined by*

$$P(x, y) = \begin{cases} \dfrac{1}{deg(x)} & \text{if } (x, y) \in E \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

So when at vertex x, the probability of choosing to move to any neighbor is the same.

We denote $p^{(k)}$ as the distribution of the random walk at time k, we know from [67] that for every non bipartite graph G, the simple random walk on G has polynomial mixing time and tends to a unique stationary distribution.

$$\pi(x) = \frac{deg(x)}{2|E|} \qquad \forall x \in V \tag{4.11}$$

This random walk brings to us the idea of computing $\frac{\|\boldsymbol{w}^{(t)}\|_1}{m}$ with local computations, let every node sum up the $w_e^{(t)}$ of its incident edges, i.e. let $\xi_u^{(t)} = \frac{1}{2}\sum_{e \sim u} w_e^{(t)}$,

then launch the same number of simple random walks as the sum, we know the total number of random walks on this graph is equal to $\sum_{u \in V} \xi_u^{(t)} = \left\| \boldsymbol{w}^{(t)} \right\|_1$, and since the single random walk towards to a distribution of $\frac{deg(u)}{2|E|}$ for node u, the expectation of walks on node u is then $\frac{deg(u)\left\| \boldsymbol{w}^{(t)} \right\|_1}{2|E|}$, then let every node multiply this quantity by $\frac{2}{deg(u)}$, and $\frac{\left\| \boldsymbol{w}^{(t)} \right\|_1}{m}$ is computed in every node. In the implementation we don't need to use random walks since all we need is to compute the expectation so matrix multiplication is enough as in our Algorithm described in Formula 4.12 and Algorithm 9.

Weighted random walk with edge conductances

Given weights c_{uv}, $(u, v) \in E$ to $G = (V, E)$, let $C(v) = \sum_{u \in N(v)} c_{uv}$, we could obtain a random walk by setting transition probability p_{uv} from u to v into $p_{uv} = \frac{c_{uv}}{C(v)}$. Let s and t be two vertices of the graph, it is well-known that the probability of a walk starting at s arriving u before arriving t is equal to the potential of u in the electrical network with C and source potential of value 1 [33].

By this connection, we could develop a fractional token diffusion process: we deliver $C(s)$ tokens at s following the random walk defined above, and let t be an absorbing vertex: any token arriving at t will be absorbed, in [6] authors showed that the number of tokens on each node converges to a constant factor of the potential of the node. We adopted the similar process in our algorithm, and provide the proof based on [6] in Section 4.4.3.

Time-varying issue

We explained our basic blocks for solving an electrical network by processes inspired from the random walk, but another question is that we are in an iterative algorithm (cf. the procedure given in Figure 1 and 2 in [20]) and quantities are changing in each round, however the two processes need time to converge. To resolve this issue, we used two "slow down" factor ensuring that the variables does not change much before the above two processes converging, i.e. $\boldsymbol{w}^{(t)}$ does not change too much before getting a good approximation of $\frac{\left\| \boldsymbol{w}^{(t)} \right\|_1}{m}$ and $\boldsymbol{r}^{(t)}$ neither before we getting a good approximation of potentials of the electrical network.

4.3 Algorithm

In this section we present our Pure-LOCAL algorithm for the weaker decision problem of $(1 + \varepsilon)$ of Max-flow, i.e. given a graph $G = (V, E)$ with edge capacity U_e, $e \in E$ and a value F, assume F^* is the max-flow value, we will compute a vector $\bar{\boldsymbol{f}}$, such that if $F \leq F^*$ then $cong_e(\bar{\boldsymbol{f}}) \leq 1$, and if $cong_e(\bar{\boldsymbol{f}}) > \frac{(1+\varepsilon)^2}{(1-\varepsilon)^2}$ then $F > F^*$.

We first describe our algorithm in a dynamic system, where we denote $\underline{A}$ for a matrix obtained by putting the last row and column of A by zeros.

Initialization:

$$w_e^{(0)} = 1, \forall e \in E$$
$$\underline{b}_{source} = F, \underline{b}_u = 0, u \neq source$$
other variables set to 0.

Parameters:

ε : the input of approximation error requirement

$\rho = \sqrt{\frac{15m}{\varepsilon}}$

T_{intl} : initialization time, to fix in Section 4.4.3

λ, α : slow down factor, to fix in Section 4.4.3

$T_{max} = T_{intl} + \frac{2\rho \ln m}{\alpha \varepsilon^2}$

The algorithm:

$$\begin{aligned}
\mathbf{y}^{(t)} &= \mathbf{y}^{(t-1)} \underline{P}^{(t-1)} + \underline{\mathbf{b}} \\
\phi_u^{(t)} &= \frac{1}{\sum_{v' \in N(u)} c_{uv'}^{(t)}} y_u^{(t)} \\
f_{uv}^{(t)} &= \left| (\phi_u^{(t)} - \phi_v^{(t)}) c_{uv}^{(t)} \right| \\
\hat{f}_{uv}^{(t)} &= \hat{f}_{uv}^{(t-1)} + f_{uv}^{(t)} \\
cong_e^{(t)} &= \frac{|f_e^{(t)}|}{U_e} \\
w_e^{(t)} &= \left(w_e^{(t-1)}\right) \left(1 + \frac{\alpha\varepsilon}{\rho} cong_e(f^{(t-1)})\right) \text{ if } t \geq T_{intl} \\
\text{denote } \xi_u^{(t)} &= \frac{1}{2} \sum_{e \sim u} w_e^{(t)} \\
\mathbf{z}^{(t)} &= (1-\lambda)\mathbf{z}^{(t-1)} B + \lambda \xi_u^{(t-1)} \\
\tau_e^{(t)} &= \frac{2(\boldsymbol{z}^{(t)})_u}{\deg(u)}, e \in E, u \text{ any vertex incident to } e \\
r_e^{(t)} &= \frac{1}{U_e^2} (w_e^{(t)} + \frac{\varepsilon}{3} \tau_e^{(t)}) \\
c_e^{(t)} &= \frac{1}{r_e^{(t)}} \\
P_{e=uv}^{(t)} &= \frac{c_e^{(t)}}{\sum_{v' \in N(u)} c_{uv'}^{(t)}}
\end{aligned} \tag{4.12}$$

Finalization: for $t > T_{max}$, let $\bar{f}_e \leftarrow \frac{(1-\varepsilon)^2}{(1+\varepsilon)} \hat{f}_e^{(t)} / (T_{max} - T_{intl})$ if $\bar{f}_e > \frac{(1+\varepsilon)^2}{(1-\varepsilon)^2} u_e$ for some $e \in E$, return NO; otherwise return YES. The result of the algorithm is YES

if all nodes return YES, and NO if any node returns NO.

Then we describe the implementation in Pure-LOCAL model, notice that here for simplicity of understanding, we describe with two "send" and "receive" operations per round, but in fact it could be just combined in one operation each.

Here the Step 2 in Algorithm 9 (mainly $\mathbf{z}^{(t)} = (1-\lambda)\mathbf{z}^{(t-1)}B + \lambda\xi_u^{(t-1)}$ in Formula 4.12) is served for approximating $\frac{\|\boldsymbol{w}^{(t)}\|_1}{m}$ with the idea of simulating simple random walk described in Section 4.2.3, precisely, we will show in Section 4.4.3 that $\tau_e^{(t)}$ approximates $\frac{\|\boldsymbol{w}^{(t)}\|_1}{m}$.

And the Step 1 in Algorithm 9 (equivalently, $\mathbf{y}^{(t)} = \mathbf{y}^{(t-1)}\underline{P}^{(t-1)} + \underline{\mathbf{b}}$ in Formula 4.12) is for approximating the solution of the electrical network simulating the token diffusion process explained in in Section 4.2.3 and we will show in Section 4.4.3 that $\phi_u^{(t)}$ approximates the potential of u in the electrical network with resistances $\boldsymbol{r}^{(t)}$ and electrical flow value F.

The re-weighting step, i.e. the $w_e^{(t)} = \left(w_e^{(t-1)}\right)\left(1 + \frac{\alpha\varepsilon}{\rho} cong_e(f^{(t-1)})\right)$ and $r_e^{(t)} = \frac{1}{U_e^2}(w_e^{(t)} + \frac{\varepsilon}{3}\tau_e^{(t)})$ corresponds to the multiplicative weight update method by [20] described in Formula 4.5. We will show that this iteratively choose of $\boldsymbol{r}^{(t)}$ will lead to $cong_e(\bar{\boldsymbol{f}}) \leq 1$ if F does not exceed the maximum flow.

4.4 Proof

This section we present the proof of our algorithm. First we show that if we could get a good approximation of the average weight $\frac{\|\boldsymbol{\xi}^{(t)}\|_1}{m}$ and an approximation of the electrical flow such that the energy is not far from the exact electrical flow, then if if F is a good approximation we could have a vector of congestion bounded by 1.

4.4.1 Weighted-average Congestion Bounded Flow

Here we assume that we could get a good approximation of the average weight and the electrical flow and show a characterization that implies a flow value F could not be a feasible value for Max-flow. On the other hand, if F is feasible, we could ensure the weighted-average of congestions is bounded. We will show in 4.4.3 that our algorithm provides the approximation assumed in this section.

Given a flow $\boldsymbol{f}$ and resistances $\boldsymbol{r}$, denote the energy of $\boldsymbol{f}$ by $\mathcal{E}_r(\boldsymbol{f}) = \sum_{e\in E} r_e f(e)^2$.

Lemma 4.4.1. *Given a value F, let $\tilde{\boldsymbol{f}}^{(t)}$ the electrical flow associated to F and $\boldsymbol{r}^{(t)}$. Let $\boldsymbol{f}^{(t)}$ be the flow computed in our algorithm in time t. Assume the solution of Max-flow is F^*. For our system, suppose that for $\varepsilon_{en} \geq 0$ and $\delta_\tau \geq 0$, there exists T_{intl} such that $\forall t > T_{intl}$,*

(1) $(1-\varepsilon_{en})\mathcal{E}_{\boldsymbol{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)}) \leq \mathcal{E}_{\boldsymbol{r}^{(t)}}(\boldsymbol{f}^{(t)}) \leq (1+\varepsilon_{en})\mathcal{E}_{\boldsymbol{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)})$

(2) $(1-\delta_\tau)\frac{\|\boldsymbol{\xi}^{(t)}\|_1}{m} \leq \tau_e^{(t)} \leq (1+\delta_\tau)\frac{\|\boldsymbol{\xi}^{(t)}\|_1}{m}$

Algorithm 9 Approximate Flow Decision Algorithm for node u

procedure DIFFUSION(Input: F)
  if t=0 **then**
    initialize all variables to 0
    $w(u,v) \leftarrow 1$
  end if
  $\xi(u) \leftarrow \frac{1}{2}\sum_{v\in N(u)} w(u,v)$
  //Step 1
  if $id(u) ==$ source **then**
    $y(u) \leftarrow y(u) + F$
  end if
  if $id(u) ==$ sink **then**
    $y(u) \leftarrow 0$
  end if
  for every neighbor v of u **do**
    send $y(u) * P(u,v)$ to v
  end for
  for every neighbor v of u **do**
    $\delta_{uv} \leftarrow$ value received from v
    $f(u,v) \leftarrow |\delta_{uv} - y(u) * P(u,v)|$
    $f_{sum}(u,v) \leftarrow f_{sum}(u,v) + f(u,v)$
    $y(u) \leftarrow y(u) + \delta_{uv}$
    $cong_{uv} \leftarrow f(u,v)/U_{uv}$
    if $t \geq T_{intl}$ **then**
      $w(u,v) \leftarrow w(u,v)(1 + \alpha\frac{\varepsilon}{\rho}cong_{uv})$
    end if
  end for
  //Step 2
  for every neighbor v of u **do**
    send $z(u)/deg(u)$ to v
  end for
  for every neighbor v of u **do**
    $\Delta_u \leftarrow$ sum of value received from v
    $Z(u) \leftarrow (1-\lambda)\Delta_u + \lambda w(u,v)$
    $\tau_{uv} \leftarrow \frac{2(\boldsymbol{z}^{(t)})_u}{\deg(u)}$
    $r_{uv} \leftarrow \frac{1}{u_{uv}^2}(w_{uv} + \frac{\varepsilon}{3}\tau_{uv})$
    $c_{uv} \leftarrow \frac{1}{r_{uv}}$
  end for
  $P(u,v) \leftarrow \frac{c_{uv}}{\sum_{v\in N(u)} c_{uv}}$
  $t \leftarrow t+1$
  if $t > T_{max}$ **then**
    $\bar{f}_{uv} \leftarrow \frac{(1-\varepsilon)^2}{(1+\varepsilon)} f_{sum}(u,v)/(T_{max} - T_{intl})$
    if $\bar{f}_{uv} > \frac{(1+\varepsilon)^2}{(1-\varepsilon)^2}U_{uv}$ **then** return NO
    else return YES
    end if
  end if
end procedure

then if $F \leq F^*$ *we have*

(a) $\sum_{e\in E} w_e^{(t)} cong_e(f^{(t)}) \leq \sqrt{(1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))}\|\boldsymbol{w}^{(t)}\|_1$ *and*

(b) $cong_e(f^{(t)}) \leq (1+\varepsilon_{en})\sqrt{\frac{1}{1-\delta_\tau}(1+\frac{\varepsilon}{3}(1+\delta_\tau))}\cdot\sqrt{\frac{3m}{\varepsilon}}$.

Proof. Assume $\boldsymbol{f}^*$ is the maximum flow with value F^*, then

$$\begin{aligned}
\mathcal{E}_{r^{(t)}}(\boldsymbol{f}^*) &= \sum_{e\in E} r_e^{(t)} f^*(e)^2 \\
&= \sum_{e\in E} (w_e^{(t)} + \frac{\varepsilon}{3}\tau_e^{(t)})\frac{f^*(e)^2}{u_e^2} \\
&= \sum_{e\in E} (w_e^{(t)} + \frac{\varepsilon}{3}\tau_e^{(t)}) cong_e^2(f^*) \\
&\leq \sum_{e\in E} (w_e^{(t)} + \frac{\varepsilon}{3}\tau_e^{(t)}) \\
&= \|\boldsymbol{w}^{(t)}\|_1 + \frac{\varepsilon}{3}\|\boldsymbol{\tau}^{(t)}\|_1
\end{aligned}$$

Because for $\forall t \geq T_{intl}$, $(1-\delta_\tau)\frac{\|\boldsymbol{\xi}^{(t)}\|_1}{m} \leq \tau_e^{(t)} \leq (1+\delta_\tau)\frac{\|\boldsymbol{\xi}^{(t)}\|_1}{m}$, then

$$\mathcal{E}_{\mathbf{r}^{(t)}}(\boldsymbol{f}^*) \leq \|\boldsymbol{w}^{(t)}\|_1 + \frac{\varepsilon}{3}(1+\delta_\tau)\left\|\boldsymbol{\xi}^{(t)}\right\|_1$$

Since $\left\|\boldsymbol{\xi}^{(t)}\right\|_1 = \|\boldsymbol{w}^{(t)}\|_1$,

$$\mathcal{E}_{\mathbf{r}^{(t)}}(\boldsymbol{f}^*) \leq (1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1$$

since given r_e, $e \in E$ the electric flow minimizes the energy among all $s-t$ flows of value F^*, this is an upper bound of the energy of any electrical flow of value $F \leq F^*$, thus $\mathcal{E}_{\mathbf{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)}) \leq (1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1$.

Thus $\mathcal{E}_{\mathbf{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)}) > (1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1$ implies $F > F^*$. Since we assumed $\mathcal{E}_{\mathbf{r}^{(t)}}(\boldsymbol{f}^{(t)}) \leq (1+\varepsilon_{en})\mathcal{E}_{\mathbf{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)})$, if $\frac{1}{(1+\varepsilon_{en})}\mathcal{E}_{\mathbf{r}^{(t)}}(\boldsymbol{f}^{(t)}) > (1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1$ then $F > F^*$.

Claim 4.4.1. *If* $\frac{1}{(1+\varepsilon_{en})}\mathcal{E}_{\boldsymbol{r}^{(t)}}(\boldsymbol{f}^{(t)}) > (1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1$ *then* $F > F^*$.

In the following we assume $F \leq F^*$ to show the property for $F \leq F^*$. We compute the energy of $\boldsymbol{f}^{(t)}$, the flow of our algorithm at time t regarding the resistance $\mathbf{r}^{(t)}$, since we have $(1-\varepsilon_{en})\mathcal{E}_{\mathbf{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)}) \leq \mathcal{E}_{\mathbf{r}^{(t)}}(\boldsymbol{f}^{(t)}) \leq (1+\varepsilon_{en})\mathcal{E}_{\mathbf{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)})$, then

$$\mathcal{E}_{\mathbf{r}^{(t)}}(\boldsymbol{f}^{(t)}) = \sum_{e\in E} r_e^{(t)} \left(f^{(t)}(e)\right)^2$$
$$\mathcal{E}_{\mathbf{r}^{(t)}}(\boldsymbol{f}^{(t)}) \leq (1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1$$

By the definition $\mathcal{E}_r(\boldsymbol{f}) = \sum_{e\in E} r_e f(e)^2$ and $r_e^{(t)} = \frac{1}{U_e^2}(w_e^{(t)} + \frac{\varepsilon}{3}\tau_e^{(t)})$, thus

$$\sum_{e\in E} \frac{1}{U_e^2}(w_e^{(t)} + \frac{\varepsilon}{3}\tau_e^{(t)}) \left(f^{(t)}(e)\right)^2 \leq (1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1$$

we have

$$\sum_{e\in E} w_e^{(t)} \left(cong_e(f^{(t)})\right)^2 \leq (1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1 \tag{4.13}$$

and

$$\sum_{e\in E} \frac{\varepsilon}{3}\tau_e^{(t)} \left(cong_e(f^{(t)})\right)^2 \leq (1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1 \tag{4.14}$$

By Cauchy-Schwarz inequality,

$$\left(\sum_{e\in E} w_e^{(t)} cong_e(f^{(t)})\right)^2 \leq \|\boldsymbol{w}^{(t)}\|_1 \left(\sum_{e\in E} w_e^{(t)} \left(cong_e(f^{(t)})\right)^2\right)$$

(4.13) implies

$$\begin{aligned}\left(\sum_{e\in E} w_e^{(t)} cong_e(f^{(t)})\right)^2 &\leq \|\boldsymbol{w}^{(t)}\|_1(1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t_0)}\|_1 \\ &\leq \|\boldsymbol{w}^{(t)}\|_1 \cdot (1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1 \\ &\leq (1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1^2\end{aligned}$$

$$\sum_{e\in E} w_e^{(t)} cong_e(f^{(t)}) \leq \sqrt{(1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))}\|\boldsymbol{w}^{(t)}\|_1$$

From Formula (4.14), we have $\forall e$,

$$\frac{\varepsilon}{3}\tau_e^{(t)}\left(cong_e(f^{(t)})\right)^2 \le (1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1$$

$$\frac{\varepsilon}{3}(1-\delta_\tau)\frac{\left\|\boldsymbol{\xi}^{(t)}\right\|_1}{m}\left(cong_e(f^{(t)})\right)^2 \le (1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1$$

$$\frac{\varepsilon}{3}(1-\delta_\tau)\frac{\left\|\boldsymbol{w}^{(t)}\right\|_1}{m}\left(cong_e(f^{(t)})\right)^2 \le (1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))\|\boldsymbol{w}^{(t)}\|_1$$

$$\left(cong_e(f^{(t)})\right)^2 \le \frac{3m}{\varepsilon}\frac{1}{(1-\delta_\tau)}(1+\varepsilon_{en})(1+\frac{\varepsilon}{3}(1+\delta_\tau))$$

$$cong_e(f^{(t)}) \le (1+\varepsilon_{en})\sqrt{\frac{1}{1-\delta_\tau}(1+\frac{\varepsilon}{3}(1+\delta_\tau))}\cdot\sqrt{\frac{3m}{\varepsilon}}$$

□

If we fix ε_{en} to be ε with $0 \le \varepsilon \le 1/2$ and δ_τ to be $\frac{1}{5}$, we could get the following corollary.

Corollary 4.4.1. *For our system, assume that for $\varepsilon_{en} \ge 0$ and $\delta_\tau \ge 0$, there exists T_{intl} such that $\forall t > T_{intl}$,*

(1) $(1-\varepsilon)\mathcal{E}_{\boldsymbol{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)}) \le \mathcal{E}_{\boldsymbol{r}^{(t)}}(\boldsymbol{f}^{(t)}) \le (1+\varepsilon)\mathcal{E}_{\boldsymbol{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)})$

(2) $(1-\frac{1}{5})\frac{\left\|\boldsymbol{\xi}^{(t)}\right\|_1}{m} \le \tau_e^{(t)} \le (1+\frac{1}{5})\frac{\left\|\boldsymbol{\xi}^{(t)}\right\|_1}{m}$

then if $\frac{1}{(1+\varepsilon)}\mathcal{E}_{\boldsymbol{r}^{(t)}}(\boldsymbol{f}^{(t)}) > (1+\frac{\varepsilon}{2})\|\boldsymbol{w}^{(t)}\|_1$ then $F > F^*$, and if $F \le F^*$ we have

(a) $\sum_{e\in E} w_e^{(t)} cong_e(f^{(t)}) \le \sqrt{(1+\varepsilon)(1+\frac{\varepsilon}{3}(1+\frac{1}{5}))}\|\boldsymbol{w}^{(t)}\|_1 \le (1+\varepsilon)\|\boldsymbol{w}^{(t)}\|_1$ *and*

(b) $cong_e(f^{(t)}) \le (1+\varepsilon)\sqrt{\frac{1}{1-\frac{1}{5}}(1+\frac{\varepsilon}{3}(1+\frac{1}{5}))}\cdot\sqrt{\frac{3m}{\varepsilon}} \le \sqrt{\frac{15m}{\varepsilon}}$.

4.4.2 Analysis of Weights Updating

Then we show that from Corollary 4.4.1, if F is feasible then we could have a vector of congestion bounded by 1.

Lemma 4.4.2. *For our system, for $\alpha, \varepsilon \ge 0$, assume there exists ρ, T_{intl}, such that*

(1) for $t \ge T_{intl}$, $\sum_e w_e^{(t)} cong_e(f^{(t)}) \le (1+\varepsilon)\|\boldsymbol{w}^{(t)}\|_1$

(2) $\alpha \cdot cong_e(f^{(t)}) \le \rho$, for all t

Then for $\bar{\boldsymbol{f}} = \frac{(1-\varepsilon)^2}{(1+\varepsilon)}\frac{1}{(N-T_{intl})}\sum_{k=T_{intl}}^{N} f^{(k)}$, if $F \le F^*$ then $cong_e(\bar{\boldsymbol{f}}) \le 1$. And if $cong_e(\bar{\boldsymbol{f}}) > \frac{(1+\varepsilon)^2}{(1-\varepsilon)^2}$ then $F > F^*$.

Proof.

$$w_e^{(t)} = \left(w_e^{(t-1)}\right)^{1-\beta}\left(1+\frac{\alpha\varepsilon}{\rho}cong_e(f^{(t-1)})\right)$$

For $t \geq T_{intl}$, we have $w_e^{(t)} = \prod_{k=T_{intl}}^{t}(1+\frac{\alpha\varepsilon}{\rho}cong_e(f^{(k)}))$, because $\alpha\cdot cong_e(f^{(k)}) \leq \rho$ and $\forall\varepsilon, x \in [0,1], (1+\varepsilon x) \geq \exp((1-\varepsilon)\varepsilon x)$ then

$$\begin{aligned} w_e^{(t)} &\geq \prod_{k=T_{intl}}^{t} \exp(\frac{(1-\varepsilon)\alpha\varepsilon}{\rho}cong_e(f^{(k)})) \\ &= \exp(\sum_{k=T_{intl}}^{t} \frac{(1-\varepsilon)\alpha\varepsilon}{\rho}cong_e(f^{(k)})) \\ &= \exp(\frac{(1-\varepsilon)\alpha\varepsilon}{\rho}\sum_{k=T_{intl}}^{t} cong_e(f^{(k)})) \end{aligned}$$

For $t = N$, let $f_e^{avg} = \frac{1}{(N-T_{intl})}\sum_{k=T_{intl}}^{N} f^{(k)}$

$$\begin{aligned} w_e^{(N)} &\geq \exp(\frac{(1-\varepsilon)\alpha\varepsilon}{\rho}\sum_{k=T_{intl}}^{N} cong_e(f^{(k)})) \\ &= \exp(\frac{(1-\varepsilon)\alpha\varepsilon(N-T_{intl})}{\rho}cong_e(f_e^{avg})) \end{aligned}$$

Then we bound $\|\boldsymbol{w}^{(N)}\|_1$

For $t > T_{intl}$,

$$\begin{aligned} \|\boldsymbol{w}^{(t)}\|_1 &= \sum_e w_e^{(t-1)}\left(1+\frac{\alpha\varepsilon}{\rho}cong_e(f^{(t-1)})\right) \\ &= \sum_e w_e^{(t-1)} + \sum_e w_e^{(t-1)}\frac{\alpha\varepsilon}{\rho}cong_e(f^{(t-1)}) \\ &= \|\boldsymbol{w}^{(t-1)}\|_1 + \sum_e w_e^{(t-1)}\frac{\alpha\varepsilon}{\rho}cong_e(f^{(t-1)}) \\ &\leq \|\boldsymbol{w}^{(t-1)}\|_1 + \frac{\alpha\varepsilon}{\rho}(1+\varepsilon)\|\boldsymbol{w}^{(t-1)}\|_1 \\ &= (1+\frac{\alpha\varepsilon}{\rho}(1+\varepsilon))\|\boldsymbol{w}^{(t-1)}\|_1 \\ &\leq \|\boldsymbol{w}^{(t-1)}\|_1 \exp\left(\frac{\alpha\varepsilon}{\rho}(1+\varepsilon)\right) \end{aligned}$$

Thus as $\|\boldsymbol{w}^{(T_{intl})}\|_1 = m$

$$\|\boldsymbol{w}^{(N)}\|_1 \leq m \exp\left((N - T_{intl})\frac{\alpha\varepsilon}{\rho}(1+\varepsilon)\right)$$

Since we have $w_e^{(t)} \leq \|\boldsymbol{w}^{(t_0)}\|_1$,

$$\exp\left(\frac{(1-\varepsilon)\alpha\varepsilon(N-T_{intl})}{\rho} cong_e(f_e^{avg})\right) \leq w_e^{(N)} \leq \|\boldsymbol{w}^{(N)}\|_1 \leq m \exp\left((N - T_{intl})\frac{\alpha\varepsilon}{\rho}(1+\varepsilon_{en})\right)$$
$$\frac{(1-\varepsilon)\alpha\varepsilon(N-T_{intl})}{\rho} cong_e(f_e^{avg}) \leq \ln m + (N - T_{intl})\frac{\alpha\varepsilon}{\rho}(1+\varepsilon)$$

Then for $N = T_{max} = T_{intl} + \frac{2\rho \ln m}{\alpha\varepsilon^2}$, $(N - T_{intl}) = \frac{2\rho \ln m}{\alpha\varepsilon^2}$

$$\begin{aligned} cong_e(f_e^{avg}) &\leq \frac{\rho}{(1-\varepsilon)\alpha\varepsilon(N-T_{intl})} \ln m + (N - T_{intl})\frac{\alpha\varepsilon}{\rho}(1+\varepsilon_{en})\frac{\rho}{(1-\varepsilon)\alpha\varepsilon(N-T_{intl})} \\ &= \frac{\rho}{(1-\varepsilon)\alpha\varepsilon(N-T_{intl})} \ln m + \frac{(1+\varepsilon)}{(1-\varepsilon)} \\ &= \frac{\varepsilon}{2(1-\varepsilon)} + \frac{(1+\varepsilon)}{(1-\varepsilon)} \end{aligned}$$

Then

$$\begin{aligned} \frac{(1-\varepsilon)^2}{(1+\varepsilon)} cong_e(f_e^{avg}) &\leq \frac{\varepsilon(1-\varepsilon)}{2(1+\varepsilon)} + (1-\varepsilon) \\ &\leq \frac{\varepsilon(1-\varepsilon)}{2(1+\varepsilon)} + (1-\varepsilon) \\ &\leq 1 \end{aligned}$$

Denote $\bar{f} = \frac{(1-\varepsilon)^2}{(1+\varepsilon)} f^{avg}$, then if $F \leq F^*$, $cong_e(\bar{f}) \leq 1 \ \forall e \in E$.

And if $cong_e(\bar{f}) > \frac{(1+\varepsilon)^2}{(1-\varepsilon)^2}$ then there must be some time k, $\sum_e w_e^{(k)} \frac{\alpha\varepsilon}{\rho} \frac{(1+\varepsilon)}{(1-\varepsilon)^2} cong_e(f^{(k)}) > (1+\varepsilon)\frac{\alpha\varepsilon}{\rho}\frac{(1+\varepsilon)^2}{(1-\varepsilon)^2}\|\boldsymbol{w}^{(k)}\|_1$, meaning

$$\frac{1}{(1+\varepsilon)}\sum_e w_e^{(k)} cong_e(f^{(k)}) > (1+\varepsilon)\|\boldsymbol{w}^{(k)}\|_1$$
$$\frac{1}{(1+\varepsilon)^2}\|\boldsymbol{w}^{(k)}\|_1\sum_e w_e^{(k)} cong_e^2(f^{(k)}) > (1+\varepsilon)^2\|\boldsymbol{w}^{(k)}\|_1^2$$
$$\frac{1}{(1+\varepsilon)^2}\sum_e w_e^{(k)} cong_e^2(f^{(k)}) > (1+\varepsilon)^2\|\boldsymbol{w}^{(k)}\|_1$$

Because $\frac{1}{(1+\varepsilon)}\mathcal{E}_{\mathbf{r}^{(t)}}(\boldsymbol{f}^{(t)}) \geq \frac{1}{(1+\varepsilon)^2}\sum_e w_e^{(k)} cong_e^2(f^{(k)})$,

$$\frac{1}{(1+\varepsilon)}\mathcal{E}_{\mathbf{r}^{(k)}}(\boldsymbol{f}^{(k)}) > (1+\varepsilon)^2\|\boldsymbol{w}^{(k)}\|$$

from Corollary 4.4.1, $F > F^*$.

□

4.4.3 Approximating the Electrical Flow

We have seen the property for F when we have a good approximation of average weight and the electrical flow, this section presents the results to approximate these quantities.

We start with the following Lemma, which follows directly from the definition of $\boldsymbol{\xi}^{(t)}$ and $\boldsymbol{\xi}^{*(t)}$.

Lemma 4.4.3. *Given $k \geq 0$ and $w_e^{(k)}$, $e \in E$, let $\xi_u^{(k)} = \frac{1}{2}\sum_{e\sim u} w_e^{(k)}$, $u \in V$ and $\xi_u^{*(k)} = \frac{deg(u)\left\|\boldsymbol{\xi}^{(t)}\right\|_1}{2m}$, $u \in V$. If for t_0, t, we know $(1-\delta_w)\boldsymbol{w}^{(t)} \leq \boldsymbol{w}^{(t_0)} \leq \boldsymbol{w}^{(t)}$ for some $\delta_w > 0$, then we have $(1-\delta_w)\boldsymbol{\xi}^{(t)} \leq \boldsymbol{\xi}^{(t_0)} \leq \boldsymbol{\xi}^{(t)}$, and $(1-\delta_w)\boldsymbol{\xi}^{*(t)} \leq \boldsymbol{\xi}^{*(t_0)} \leq \boldsymbol{\xi}^{*(t)}$.*

Analysis of $\boldsymbol{\xi}$

$\boldsymbol{\xi}$ represents the quantity that every nodes collects the sum of weights of incident edges (and divide by 2), and we are going to show that $\boldsymbol{\xi}^{(t)}B^i$ is approximating $\boldsymbol{\xi}^{*(t)}$, which is a factor of the average of weights.

Lemma 4.4.4. *Let $\boldsymbol{\xi}^{(t)}$ and B be defined as in our system 4.12, denote*

$$\xi_u^{*(t)} = \frac{deg(u)\left\|\boldsymbol{\xi}^{(t)}\right\|_1}{2m}$$

given $\delta \geq 0$, let $mix_\delta(P)$ be the δ-mixing time of the simple random walk on G as in Definition 4.2.4, then for $i \geq mix_\delta(P)$, we have

$$\left\|\boldsymbol{\xi}^{(t)}B^i - \boldsymbol{\xi}^{*(t)}\right\|_\infty \leq \delta.$$

Proof. This follows from the simple random walk on G and its mixing time.

Consider the simple random walk on G starting with initial distribution

$$\pi^{(0)} = \frac{\boldsymbol{\xi}^{(t)}}{\left\|\boldsymbol{\xi}^{(t)}\right\|_1}$$

since B is equal to the transition matrix of simple random walk in Definition 4.2.4, according to the definition of random walk, the distribution of this random walk at time i is

$$\pi^{(i)} = \frac{\boldsymbol{\xi}^{(t)} B^i}{\left\|\boldsymbol{\xi}^{(t)}\right\|_1}$$

Then according to the definition of δ-mixing time in Definition 4.2.3, we have that the stationary distribution of this random walk is $\pi_u = \frac{deg(u)}{2m}$, then for $i \geq mix_\delta(P)$,

$$\left\|\pi^{(i)} - \pi\right\|_\infty = \max_{u \in V} \left(\frac{(\boldsymbol{\xi}^{(t)} B^i)_u}{\left\|\boldsymbol{\xi}^{(t)}\right\|_1} - \frac{deg(u)}{2m} \right) \leq \delta$$

then

$$\begin{aligned}
\left\|\boldsymbol{\xi}^{(t)} B^i - \boldsymbol{\xi}^{*(t)}\right\|_\infty &= \max_{u \in V} \left(\left(\boldsymbol{\xi}^{(t)} B^i\right)_u - \frac{deg(u) \left\|\boldsymbol{\xi}^{(t)}\right\|_1}{2m} \right) \\
&= \left\|\boldsymbol{\xi}^{(t)}\right\|_1 \max_{u \in V} \left(\frac{(\boldsymbol{\xi}^{(t)} B^i)_u}{\left\|\boldsymbol{\xi}^{(t)}\right\|_1} - \frac{deg(u)}{2m} \right) \\
&\leq \delta \left\|\boldsymbol{\xi}^{(t)}\right\|_1
\end{aligned}$$

□

From Proposition 4.2.1 we know that the error parameter brings a logarithmic factor to the mixing time, and we could have the following corollary.

Corollary 4.4.2. *Let $\boldsymbol{\xi}^{(t)}$, $\boldsymbol{\xi}^{*(t)}$ and B be defined as above, let $mix(B) = mix_{1/3}(B)$ then for any $0 < \delta_\xi < \frac{1}{2}$, for all $i \geq \lceil \log_2 (\frac{2m}{\delta_\xi}) \rceil \cdot mix(B)$,*

$$\left\|\boldsymbol{\xi}^{(t)} B^i - \boldsymbol{\xi}^{*(t)}\right\|_\infty \leq \delta_\xi \frac{\left\|\boldsymbol{\xi}^{(t)}\right\|_1}{2m}.$$

and

$$\left|\boldsymbol{\xi}^{(t)} B^i - \boldsymbol{\xi}^{*(t)}\right| \leq \delta_\xi \cdot \boldsymbol{\xi}^{*(t)}.$$

Analysis of z

Then we analyze $\mathbf{z}^{(t+1)} = (1-\lambda)\mathbf{z}^{(t)} B + \lambda \xi_u^{(t)}$ in our system 4.12. $\mathbf{z}^{(t)}$ is the variable approximating the average of weights with varying weights, here we we assume that weights does not vary a lot (within a $(1+\delta_w)$ factor) before $\mathbf{z}$ being a good approximation. λ is the factor to make the ancient and very recent weights be small enough in iterations so that $\mathbf{z}^{(t)}$ counts only the portion that is a good approximation.

Lemma 4.4.5. *For our system, given $\delta_w, \delta_\xi \geq 0$, let $t_{mix(B)} = \lceil \log_2 (\frac{2m}{\delta_\xi}) \rceil \cdot mix(B)$, $T_z = (2m \cdot \frac{-\ln \delta_\xi}{\delta_\xi} + 1) \cdot t_{mix(B)}$, let $\lambda = 1 - \delta_\xi^{\frac{1}{T_z}}$. Assume $\forall t \geq T_z + t_{mix(B)}$, $\forall t - T_z - 1 \leq t_0 \leq t$, we have $(1-\delta_w)\boldsymbol{w}^{(t)} \leq \boldsymbol{w}^{(t_0)} \leq \boldsymbol{w}^{(t)}$, then $\forall t \geq T_z + t_{mix(B)}$,*

$$(1-3\delta_\xi)(1-\delta_w)\boldsymbol{\xi}^{*(t)} \leq \boldsymbol{z}^{(t)} \leq (1+5\delta_\xi)\boldsymbol{\xi}^{*(t)}.$$

Proof. Firstly we remind that from Corollary 4.4.2 we know for $i \geq t_{mix(B)}$, there must be $\left|\boldsymbol{\xi}^{(t)} B^i - \boldsymbol{\xi}^{*(t)}\right| \leq \delta_\xi \cdot \boldsymbol{\xi}^{*(t)}$.

By recursively developping $\mathbf{z}$ we get:

$$\begin{aligned}\mathbf{z}^{(t)} &= (1-\lambda)\mathbf{z}^{(t-1)}B + \lambda\boldsymbol{\xi}^{(t-1)} \\ &= (1-\lambda)^t \mathbf{z}^{(0)} B^t + \lambda \sum_{k=0}^{t-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1}\end{aligned}$$

since $\mathbf{z}^{(0)} = \mathbf{0}$,

$$\begin{aligned}\mathbf{z}^{(t)} =& \lambda \sum_{k=0}^{t-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1} \\ =& \lambda \sum_{k=0}^{t_0-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1} + \lambda \sum_{k=t_0}^{t-t_{mix(B)}-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1} + \\ & \lambda \sum_{k=t-t_{mix(B)}}^{t-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1}\end{aligned}$$

Denote $t_0 = t - T_z - 1$,

$$\begin{aligned}SUM_1 &= \lambda \sum_{k=0}^{t_0-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1} \\ SUM_2 &= \lambda \sum_{k=t_0}^{t-t_{mix(B)}-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1} \\ SUM_3 &= \lambda \sum_{k=t-t_{mix(B)}}^{t-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1}\end{aligned}$$

Then $\mathbf{z}^{(t)} = SUM_1 + SUM_2 + SUM_3$ and we analyze each sum here.

(1) In the sum of $SUM_1 = \lambda \sum_{k=0}^{t_0-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1}$, always $t-k-1 \geq t_{mix(B)}$, then from Lemma 4.4.4 and since $\xi_u^{*(t)}$ increases along t, we have:

$$
\begin{aligned}
SUM_1 &= \lambda \sum_{k=0}^{t_0-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1} \\
&\leq \lambda \sum_{k=0}^{t_0-1} (1-\lambda)^{t-k-1} (1+\delta_\xi) \boldsymbol{\xi}^{*(k)} \\
&= (1+\delta_\xi) \lambda \sum_{k=0}^{t_0-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{*(t_0)} \\
&= (1+\delta_\xi) \cdot \lambda \cdot (1-\lambda)^{t-t_0} \cdot \frac{1-(1-\lambda)^{t_0}}{1-(1-\lambda)} \cdot \boldsymbol{\xi}^{*(t_0)} \\
&\leq (1+\delta_\xi) \cdot \lambda \cdot (1-\lambda)^{t-t_0} \cdot \frac{1}{\lambda} \cdot \boldsymbol{\xi}^{*(t_0)} \\
&= (1+\delta_\xi)(1-\lambda)^{t-t_0} \boldsymbol{\xi}^{*(t_0)}
\end{aligned}
$$

because $t - t_0 = T_z + 1$, then for $\lambda \geq 1 - \delta_\xi^{\frac{1}{T_z}}$ we have $(1-\lambda)^{t-t_0} \leq \delta_\xi$ and

$$
\begin{aligned}
SUM_1 &\leq (1+\delta_\xi)(1-\lambda)^{t-t_0-1} \boldsymbol{\xi}^{*(t_0)} \\
&\leq \delta_\xi (1+\delta_\xi) \boldsymbol{\xi}^{*(t_0)}
\end{aligned}
\tag{4.15}
$$

(2) For the third sum $SUM_3 = \lambda \sum_{k=t-t_{mix(B)}}^{t-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1}$, note that B is a stochastic matrix and then $\boldsymbol{\xi}^{(k)} B^i \leq \left\| \boldsymbol{\xi}^{(k)} \right\|_1 \cdot \mathbf{1}$ for all $i > 0$, and $\xi_u^{*(t)}$ increases along t, then

$$
\begin{aligned}
SUM_3 &\leq \lambda \sum_{k=t-t_{mix(B)}}^{t-1} (1-\lambda)^{t-k-1} \left\| \boldsymbol{\xi}^{(t)} \right\|_1 \cdot \mathbf{1} \\
&\leq \lambda \frac{1-(1-\lambda)^{t_{mix(B)}}}{1-(1-\lambda)} \left\| \boldsymbol{\xi}^{(t)} \right\|_1 \cdot \mathbf{1} \\
&\leq \left(1-(1-\lambda)^{t_{mix(B)}}\right) \left\| \boldsymbol{\xi}^{(t)} \right\|_1 \cdot \mathbf{1}
\end{aligned}
\tag{4.16}
$$

For $\lambda = 1 - \delta_\xi^{\frac{1}{T_z}}$, $T_z = (2m \cdot \frac{-\ln \delta_\xi}{\delta_\xi} + 1) \cdot t_{mix(B)}$, since $\ln(1-x) \leq -x$ for $0 < x < 1$, we know that $-\frac{\delta_\xi}{2m} \geq \ln\left(1 - \frac{\delta_\xi}{2m}\right)$ and $-\frac{n \ln(\delta_\xi)}{\delta_\xi} \geq \frac{\ln(\delta_\xi)}{\ln(1-\frac{\delta_\xi}{2m})}$ then we have

$$
T_z \geq \frac{\ln(\delta_\xi)}{\ln\left(1-\frac{\delta_\xi}{2m}\right)} \cdot t_{mix(B)}
$$

Then

$$
\begin{aligned}
t_{mix(B)} \ln(\delta_\xi) &\geq T_z \cdot \ln\left(1 - \frac{\delta_\xi}{2m}\right) \\
1 - \delta_\xi^{\frac{1}{T_z}} &\leq 1 - \left(1 - \frac{\delta_\xi}{2m}\right)^{\frac{1}{t_{mix(B)}}}
\end{aligned}
$$

Notice that $\lambda = 1 - \delta_{\xi}^{\frac{1}{T_z}}$,

$$\begin{aligned} \lambda &\leq 1 - \left(1 - \frac{\delta_\xi}{2m}\right)^{\frac{1}{t_{mix(B)}}} \\ (1-\lambda) &\geq \left(1 - \frac{\delta_\xi}{2m}\right)^{\frac{1}{t_{mix(B)}}} \\ (1-\lambda)^{t_{mix(B)}} &\geq 1 - \frac{\delta_\xi}{2m} \end{aligned} \tag{4.17}$$

then

$$\left(1 - (1-\lambda)^{t_{mix(B)}}\right) \cdot 2m \leq \delta_\xi \tag{4.18}$$

since $\min_u deg(u) \geq 1$, then

$$\left(1 - (1-\lambda)^{t_{mix(B)}}\right) \cdot \frac{2m}{\min_u deg(u)} \leq \delta_\xi \tag{4.19}$$

From Formula 4.16 we know that

$$SUM_3 \leq \left(1 - (1-\lambda)^{t_{mix(B)}}\right) \left\|\boldsymbol{\xi}^{(t)}\right\|_1 \cdot \mathbf{1} \tag{4.20}$$

and by definition of $\xi_u^{*(t)} = \frac{deg(u)\left\|\boldsymbol{\xi}^{(t)}\right\|_1}{2m}$, we know $\left\|\boldsymbol{\xi}^{(t)}\right\|_1 \cdot \mathbf{1} \leq \frac{2m}{\min_u deg(u)} \boldsymbol{\xi}^{*(t)}$.

Combine Formula 4.19 and Formula 4.20 we get

$$SUM_3 \leq \delta_\xi \cdot \boldsymbol{\xi}^{*(t)} \tag{4.21}$$

(3) For the sum $SUM_2 = \lambda \sum_{k=t_0}^{t-t_{mix(B)}-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1}$, we have also that $t-k-1 \geq mix(B)$ and $\boldsymbol{\xi}^{*(k)}$ increases along with k, then

$$\lambda \sum_{k=t_0}^{t-t_{mix(B)}-1} (1-\lambda)^{t-k-1}(1-\delta_\xi)\boldsymbol{\xi}^{*(k)} \leq SUM_2 \leq \lambda \sum_{k=t_0}^{t-t_{mix(B)}-1} (1-\lambda)^{t-k-1}(1+\delta_\xi)\boldsymbol{\xi}^{*(k)}$$

$$(1-\delta_\xi)\lambda \sum_{k=t_0}^{t-t_{mix(B)}-1} (1-\lambda)^{t-k-1}\boldsymbol{\xi}^{*(t_0)} \leq SUM_2 \leq (1+\delta_\xi)\lambda \sum_{k=t_0}^{t-t_{mix(B)}-1} (1-\lambda)^{t-k-1}\boldsymbol{\xi}^{*(t)}$$

From Formula 4.18 we know $\lambda \sum_{k=t-t_{mix(B)}}^{t-1}(1-\lambda)^{t-k-1} = (1-(1-\lambda)^{t_{mix(B)}}) \leq \frac{\delta_\xi}{2m}$, and $\lambda \sum_{k=t_0}^{t-1}(1-\lambda)^{t-k-1} - \delta_\xi \leq \lambda \sum_{k=t_0}^{t-t_{mix(B)}-1}(1-\lambda)^{t-k-1}$, then we have

$$(1-\delta_\xi)\left(\lambda\sum_{k=t_0}^{t-1}(1-\lambda)^{t-k-1}-\delta_\xi\right)\boldsymbol{\xi}^{*(t_0)} \leq SUM_2 \leq (1+\delta_\xi)\left(\lambda\sum_{k=t_0}^{t-1}(1-\lambda)^{t-k-1}\right)\boldsymbol{\xi}^{*(t)}$$
$$(1-\delta_\xi)\left(1-(1-\lambda)^{t-t_0-1}-\delta_\xi\right)\boldsymbol{\xi}^{*(t_0)} \leq SUM_2 \leq (1+\delta_\xi)(1-(1-\lambda)^{t-t_0-1})\boldsymbol{\xi}^{*(t)}$$

since $\lambda = 1-\delta_\xi^{\frac{1}{T_z}}$, $(1-\lambda)^{t-t_0-1} \leq \delta_\xi$ then

$$\begin{aligned}(1-\delta_\xi)(1-\delta_\xi-\delta_\xi)\boldsymbol{\xi}^{*(t_0)} &\leq SUM_2 \leq (1+\delta_\xi)\boldsymbol{\xi}^{*(t)}\\ (1-\delta_\xi)(1-2\delta_\xi)\boldsymbol{\xi}^{*(t_0)} &\leq SUM_2 \leq (1+\delta_\xi)\boldsymbol{\xi}^{*(t)}\end{aligned} \tag{4.22}$$

Combine Formula 4.15, Formula 4.22 and Formula 4.21, for $\mathbf{z}^{(t)} = SUM_1 + SUM_2 + SUM_3$,

$$\begin{aligned}0+(1-\delta_\xi)(1-2\delta_\xi)\boldsymbol{\xi}^{*(t_0)}+0 \leq \mathbf{z}^{(t)} &\leq (1+\delta_\xi)\delta_\xi\boldsymbol{\xi}^{*(t_0)}+(1+\delta_\xi)\boldsymbol{\xi}^{*(t)}+\delta_\xi\boldsymbol{\xi}^{*(t)}\\ (1-\delta_\xi)(1-2\delta_\xi)\boldsymbol{\xi}^{*(t_0)} \leq \mathbf{z}^{(t)} &\leq (1+\delta_\xi)\delta_\xi\boldsymbol{\xi}^{*(t_0)}+(1+\delta_\xi)\boldsymbol{\xi}^{*(t)}+\delta_\xi\boldsymbol{\xi}^{*(t)}\end{aligned}$$

since $\xi_u^{(t)} = \frac{1}{2}\sum_{e\sim u} w_e^{(t)}$ and $(1-\delta_w)\boldsymbol{w}^{(t)} \leq \boldsymbol{w}^{(t_0)} \leq \boldsymbol{w}^{(t)}$

we have $(1-\delta_w)\boldsymbol{\xi}^{(t)} \leq \boldsymbol{\xi}^{(t_0)} \leq \boldsymbol{\xi}^{(t)}$, and since $\xi_u^{*(i)} = \frac{deg(u)\left\|\boldsymbol{\xi}^{(i)}\right\|_1}{2m}$ then $(1-\delta_w)\boldsymbol{\xi}^{*(t)} \leq \boldsymbol{\xi}^{*(t_0)} \leq \boldsymbol{\xi}^{*(t)}$, then by changing $\boldsymbol{\xi}^{*(t_0)}$ to $\boldsymbol{\xi}^{*(t)}$ in the above inequalities, we get:

$$\begin{aligned}(1-\delta_\xi)(1-2\delta_\xi)\boldsymbol{\xi}^{*(t_0)} \leq \mathbf{z}^{(t)} &\leq (1+\delta_\xi)\delta_\xi\boldsymbol{\xi}^{*(t_0)}+(1+\delta_\xi)\boldsymbol{\xi}^{*(t)}+\delta_\xi\boldsymbol{\xi}^{*(t)}\\ (1-\delta_\xi)(1-2\delta_\xi)(1-\delta_w)\boldsymbol{\xi}^{*(t)} \leq \mathbf{z}^{(t)} &\leq (1+\delta_\xi)\delta_\xi\boldsymbol{\xi}^{*(t)}+(1+\delta_\xi)\boldsymbol{\xi}^{*(t)}+\delta_\xi\boldsymbol{\xi}^{*(t)}\\ (1-3\delta_\xi)(1-\delta_w)\boldsymbol{\xi}^{*(t)} \leq \mathbf{z}^{(t)} &\leq (1+5\delta_\xi)\boldsymbol{\xi}^{*(t)}\end{aligned}$$

□

We set $\delta_\xi = \frac{1}{30}$ we could get the following corollary:

Corollary 4.4.3. *For our system, given $0 \leq \delta_w \leq \frac{1}{2}$, let $t_{mix(B)} = \lceil \log_2(60m) \rceil \cdot mix(B)$, $T_z = (2\ln(30m)+1)\cdot t_{mix(B)}$. If $\forall t \geq T_z + t_{mix(B)}$, $\forall t - T_z - 1 \leq t_0 \leq t$, we have $(1-\delta_w)\boldsymbol{w}^{(t)} \leq \boldsymbol{w}^{(t_0)} \leq \boldsymbol{w}^{(t)}$, then $\forall t - T_z \leq t_0 \leq t$,*

$$(1-\frac{1}{5})\boldsymbol{\xi}^{*(t)} \leq \boldsymbol{z}^{(t)} \leq (1+\frac{1}{5})\boldsymbol{\xi}^{*(t)}.$$

then because $\xi_u^{*(t)} = \frac{deg(u)\left\|\boldsymbol{\xi}^{(t)}\right\|_1}{2m}$, for $\tau_e^{(t)} = \frac{2z_u^{(t)}}{\deg(u)}$, $e \in E$, u any vertex incident to e, we have

$$(1-\frac{1}{5})\frac{\left\|\boldsymbol{\xi}^{(i)}\right\|_1}{m} \leq \tau_e^{(t)} \leq (1+\frac{1}{5})\frac{\left\|\boldsymbol{\xi}^{(i)}\right\|_1}{m}.$$

Lemma 4.4.6. *For our system, given* $0 \leq \delta_\xi, \delta_w \leq \frac{1}{2}$, *let* $t_{mix(B)} = \lceil \log_2 (\frac{2m}{\delta_\xi}) \rceil \cdot mix(B)$, $T_z = (2m \cdot \frac{-\ln \delta_\xi}{\delta_\xi} + 1) \cdot t_{mix(B)}$. *Assume there is* $T_{conv} > T_z$, $\forall t_0 \geq T_r = \frac{\ln\left(\frac{1}{12m}\delta_w\right)}{\ln \delta_\xi} T_z$, $\forall t_{now}$ *that* $\forall t_{now} - T_{conv} \leq t_0 \leq t_{now}$, *we have* $(1-\delta_w)\boldsymbol{w}^{(t_{now})} \leq \boldsymbol{w}^{(t_0)} \leq \boldsymbol{w}^{(t)}$, *and let* $\lambda = 1 - \delta_\xi^{\frac{1}{T_z}}$ *then* $\forall t_0 \geq T_r = \frac{\ln\left(\frac{1}{12m}\delta_w\right)}{\ln \delta_\xi} T_z$, $\forall t_{now} - T_{conv} \leq t_0 \leq t_{now}$,

$$(1-2\delta_w)\boldsymbol{\xi}^{*(t_{now})} \leq \boldsymbol{z}^{(t_0)} \leq (1+2\delta_w)\boldsymbol{\xi}^{*(t_{now})}$$

Proof. Given t_{now}, t_0, for $\mathbf{z}^{(t)} = (1-\lambda)\mathbf{z}^{(t-1)}B + \lambda\boldsymbol{\xi}^{(t-1)}$, since $\mathbf{z}^{(0)} = \mathbf{0}$

$$\begin{aligned}
\mathbf{z}^{(t)} &= (1-\lambda)^t \mathbf{z}^{(0)} B^t + \lambda \sum_{k=0}^{t-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1} \\
&= \lambda \sum_{k=0}^{t-1} (1-\lambda)^{t-k-1} \boldsymbol{\xi}^{(k)} B^{t-k-1} \\
&= \lambda \sum_{k=0}^{t} (1-\lambda)^{k} \boldsymbol{\xi}^{(t-k-1)} B^{k}
\end{aligned}$$

$$\begin{aligned}
\mathbf{z}^{(t_{now})} &= \lambda \sum_{k=0}^{t_{now}} (1-\lambda)^{k} \boldsymbol{\xi}^{(t_{now}-k-1)} B^{k} \\
&= \lambda \sum_{k=0}^{t_0} (1-\lambda)^{k} \boldsymbol{\xi}^{(t_{now}-k-1)} B^{k} + \lambda \sum_{k=t_0+1}^{t_{now}} (1-\lambda)^{k} \boldsymbol{\xi}^{(t_{now}-k-1)} B^{k}
\end{aligned}$$

$$\mathbf{z}^{(t_0)} = \lambda \sum_{k=0}^{t_0} (1-\lambda)^{k} \boldsymbol{\xi}^{(t_0-k-1)} B^{k}$$

$$\begin{aligned}
\mathbf{z}^{(t_{now})} - \mathbf{z}^{(t_0)} &= \lambda \sum_{k=0}^{t_0} (1-\lambda)^{k} \left(\boldsymbol{\xi}^{(t_{now}-k-1)} B^{k} - \boldsymbol{\xi}^{(t_0-k-1)} B^{k}\right) + \lambda \sum_{k=t_0+1}^{t_{now}} (1-\lambda)^{k} \boldsymbol{\xi}^{(t_{now}-k-1)} B^{k} \\
&= \lambda \sum_{k=0}^{t_0} (1-\lambda)^{k} \left(\boldsymbol{\xi}^{(t_{now}-k-1)} B^{k} - \boldsymbol{\xi}^{(t_0-k-1)} B^{k}\right) + \lambda \sum_{k=t_0+1}^{t_{now}} (1-\lambda)^{k} \boldsymbol{\xi}^{(t_{now}-k-1)} B^{k} \\
&= \lambda \sum_{k=0}^{t_0} (1-\lambda)^{k} \left(\boldsymbol{\xi}^{(t_{now}-k-1)} - \boldsymbol{\xi}^{(t_0-k-1)}\right) B^{k} + \lambda \sum_{k=t_0+1}^{t_{now}} (1-\lambda)^{k} \boldsymbol{\xi}^{(t_{now}-k-1)} B^{k}
\end{aligned}$$

$(t_{now} - k - 1) - T_{intl} \leq (t_0 - k - 1) \leq (t_{now} - k - 1)$, then $(1-\delta_w)\boldsymbol{\xi}^{*(t_{now}-k-1)} \leq \boldsymbol{\xi}^{*(t_0-k-1)} \leq \boldsymbol{\xi}^{*(t_{now}-k-1)}$,

$$0 \leq \boldsymbol{\xi}^{(t_{now}-k-1)} - \boldsymbol{\xi}^{(t_0-k-1)} \leq \delta_w \boldsymbol{\xi}^{*(t_{now}-k-1)}$$

Then

$$\lambda \sum_{k=0}^{t_0} (1-\lambda)^k \left(\boldsymbol{\xi}^{(t_{now}-k-1)} B^k - \boldsymbol{\xi}^{(t_0-k-1)} B^k \right) \leq \delta_w \mathbf{z}^{(t_{now})} \tag{4.23}$$

$$\lambda \sum_{k=t_0+1}^{t_{now}} (1-\lambda)^k \leq (1-\lambda)^{t_0}$$

Because $\lambda = 1 - \delta_\xi^{\frac{1}{T_z}}$,

$$\lambda \sum_{k=t_0+1}^{t_{now}} (1-\lambda)^k \leq (\delta_\xi^{\frac{t_0}{T_z}})$$

If

$$\frac{t_0}{T_z} \geq \frac{\ln\left(\frac{1}{12m}\delta_w\right)}{\ln \delta_\xi}$$

then

$$\lambda \sum_{k=t_0+1}^{t_{now}} (1-\lambda)^k \leq \frac{1}{12m}\delta_w$$

Because b is a stochastic matrix and $\left\|\boldsymbol{\xi}^{(t)}\right\|_1$ increasing along time t, then $\left\|\boldsymbol{\xi}^{(t_{now}-k-1)} B^k\right\|_1 \leq \left\|\boldsymbol{\xi}^{(t_{now}-k-1)}\right\|_1 \leq \left\|\boldsymbol{\xi}^{(t_{now})}\right\|_1$.

$$\lambda \sum_{k=t_0+1}^{t_{now}} (1-\lambda)^k \boldsymbol{\xi}^{(t_{now}-k-1)} B^k \leq \frac{1}{12m}\delta_w \left\|\boldsymbol{\xi}^{(t_{now})}\right\|_1 \cdot \mathbf{1}$$

We know from Lemma 4.4.5, that for $t_{now} \geq T_z$, $(1-3\delta_\xi)(1-\delta_w)\boldsymbol{\xi}^{*(t)} \leq \mathbf{z}^{(t)} \leq (1+5\delta_\xi)\boldsymbol{\xi}^{*(t)}$, and $\xi_u^{*(t)} = \frac{deg(u)\left\|\boldsymbol{\xi}^{(t)}\right\|_1}{2m}$, and since $\delta_\xi, \delta_w \leq \frac{1}{2}$ we know

$$\left\|\boldsymbol{\xi}^{(t_{now})}\right\|_1 \leq \frac{2m}{(1-3\delta_\xi)(1-\delta_w)} \mathbf{z}^{(t_{now})} \left\|\boldsymbol{\xi}^{(t_{now})}\right\|_1 \leq 12m\mathbf{z}^{(t_{now})}$$

Then for

$$t_0 \geq \frac{\ln\left(\frac{1}{12m}\delta_w\right)}{\ln \delta_\xi} T_z$$

$$\lambda \sum_{k=t_0+1}^{t_{now}} (1-\lambda)^k \boldsymbol{\xi}^{(t_{now}-k-1)} B^k \leq \delta_w \mathbf{z}^{(t_{now})}$$

Combine Formula 4.23 and 4.4.3, we get

$$0 \leq \mathbf{z}^{(t_{now})} - \mathbf{z}^{(t_0)} \leq 2\delta_w \mathbf{z}^{(t_{now})}$$

It means

$$(1-2\delta_w)\mathbf{z}^{(t_{now})} \le \mathbf{z}^{(t_0)} \le (1+2\delta_w)\mathbf{z}^{(t_{now})}$$

□

We set $\delta_\xi = \frac{1}{30}$ we could get the following corollary:

Corollary 4.4.4. *For our system, given* $0 \le \delta_\xi, \delta_w \le \frac{1}{2}$, *let* $t_{mix(B)} = \lceil \log_2(60m) \rceil \cdot mix(B)$, $T_z = (2\ln(30m)+1) \cdot t_{mix(B)}$,

if there is $T_{conv} \ge T_z$, $\forall t_0 \ge T_r = \frac{\ln\left(\frac{1}{12m}\delta_w\right)}{\ln \delta_\xi} T_z$, $\forall t_{now}$ *that* $\forall t_{now} - T_{conv} \le t_0 \le t_{now}$, *we have* $(1-\delta_w)\boldsymbol{w}^{(t_{now})} \le \boldsymbol{w}^{(t_0)} \le \boldsymbol{w}^{(t)}$, *then* $\forall t_0 \ge T_r$, $\forall t_{now} - T_{conv} \le t_0 \le t_{now}$,

$$(1-2\delta_w)\boldsymbol{z}^{(t_{now})} \le \boldsymbol{z}^{(t_0)} \le (1+2\delta_w)\boldsymbol{z}^{(t_{now})}$$

Analysis of r

$r_e^{(t)}$ is the resistance of edge $e \in E$ in time t.

Lemma 4.4.7. *For our system, assume for* $\delta_w, \delta_z \ge 0$, *there exists* T_{intl} *and* $T_{conv} \ge T_{intl}$ *such that* $\forall t > T_{intl}$, *there is*

(1) $(1-\delta_z)\boldsymbol{\xi}^{*(t)} \le \boldsymbol{z}^{(t)} \le (1+\delta_z)\boldsymbol{\xi}^{*(t)}$ *and*

(2) $\forall k$, $t \le k \le t + T_{conv}$, *we have* $(1-\delta_z)\boldsymbol{w}^{(k)} \le \boldsymbol{w}^{(t)} \le \boldsymbol{w}^{(k)}$.

Then for any $\forall t_0 \ge T_{intl}$, $\forall t_{now}$ *that* $t_{now} - T_{conv} \le t_0 \le t_{now}$,

$$\frac{(1-\delta_z)^2}{(1+\delta_z)^2} r_e^{(t_{now})} \le r_e^{(t_0)} \le \frac{(1+\delta_z)^2}{(1-\delta_z)^2} r_e^{(t_{now})}, \forall e \in E.$$

Proof. Firstly we prove a small fact: for $t_{now} \ge T_{intl} + T_{conv}$, $\forall t_0$ that $t_{now} - T \le t_0 \le t_{now}$, there is

$$\frac{(1-\delta_z)}{(1+\delta_z)}\mathbf{z}^{(t_0)} \le \mathbf{z}^{(t_{now})} \le \frac{(1+\delta_z)}{(1-\delta_z)^2}\mathbf{z}^{(t_0)}. \tag{4.24}$$

From $t_{now} - T_{conv} \le t_0 \le t_{now}$ we know $t_0 \le t_{now} \le t_0 + T_{conv}$, then by the assumption, $(1-\delta_z)\boldsymbol{w}^{(t_{now})} \le \boldsymbol{w}^{(t_0)} \le \boldsymbol{w}^{(t_{now})}$.

since $t_0 \ge t_{now} - T_{conv} \ge T_{intl}$, by assumption there is

$$(1-\delta_z)\boldsymbol{\xi}^{*(t_0)} \le \mathbf{z}^{(t_0)} \le (1+\delta_z)\boldsymbol{\xi}^{*(t_0)}. \tag{4.25}$$

By Lemma 4.4.3 we know $(1-\delta_z)\boldsymbol{\xi}^{*(t_{now})} \le \boldsymbol{\xi}^{*(t_0)} \le \boldsymbol{\xi}^{*(t_{now})}$

$$(1-\delta_z)^2\boldsymbol{\xi}^{*(t_{now})} \le \mathbf{z}^{(t_0)} \le (1+\delta_z)\boldsymbol{\xi}^{*(t_{now})}. \tag{4.26}$$

since $t_{now} \ge T_{intl}$, by assumption there is

$$(1-\delta_z)\boldsymbol{\xi}^{*(t_{now})} \le \mathbf{z}^{(t_{now})} \le (1+\delta_z)\boldsymbol{\xi}^{*(t_{now})}. \tag{4.27}$$

By Formula 4.26,

$$\frac{(1-\delta_z)^2}{(1+\delta_z)}\mathbf{z}^{(t_{now})} \leq \mathbf{z}^{(t_0)} \leq \frac{(1+\delta_z)}{(1-\delta_z)}\mathbf{z}^{(t_{now})} \tag{4.28}$$

Then we bound $\tau_e^{(t)}$, because in the definition $\tau_e^{(t)} = \frac{2z_u^{(t)}}{\deg(u)}$, $e \in E$, u any vertex incident to e,

$$\begin{aligned}
\frac{(1-\delta_z)^2}{(1+\delta_z)}\frac{2z_u^{(t_{now})}}{\deg(u)} &\leq \tau_e^{(t_0)} \leq \frac{(1+\delta_z)}{(1-\delta_z)}\frac{2z_u^{(t_{now})}}{\deg(u)} \\
\frac{(1-\delta_z)^2}{(1+\delta_z)}\tau_e^{(t_{now})} &\leq \tau_e^{(t_0)} \leq \frac{(1+\delta_z)}{(1-\delta_z)}\tau_e^{(t_{now})} \\
\frac{(1-\delta_z)^2}{(1+\delta_z)}\tau_e^{(t_{now})} &\leq \tau_e^{(t_0)} \leq \frac{(1+\delta_z)^2}{(1-\delta_z)}\tau_e^{(t_{now})}
\end{aligned}$$

then

$$\begin{aligned}
r_e^{(t_0)} &= \frac{1}{U_e^2}(w_e^{(t_0)} + \frac{\varepsilon}{3}\tau_e^{(t_0)}) \\
\frac{1}{U_e^2}\left((1-\delta_z)w_e^{(t_{now})} + \frac{\varepsilon}{3}\frac{(1-\delta_z)^2}{(1+\delta_z)}\tau_e^{(t_{now})}\right) &\leq r_e^{(t_0)} \leq \frac{1}{U_e^2}\left(w_e^{(t_{now})} + \frac{\varepsilon}{3}\frac{(1+\delta_z)^2}{(1-\delta_z)}\tau_e^{(t_{now})}\right) \\
\frac{1}{U_e^2}\frac{(1-\delta_z)^2}{(1+\delta_z)^2}\left(w_e^{(t_0)} + \frac{\varepsilon}{3}\tau_e^{(t_{now})}\right) &\leq r_e^{(t_{now})} \leq \frac{1}{U_e^2}\frac{(1+\delta_z)^2}{(1-\delta_z)^2}\left(w_e^{(t_{now})} + \frac{\varepsilon}{3}\tau_e^{(t_{now})}\right)
\end{aligned}$$

equivalently,

$$\frac{(1-\delta_z)^2}{(1+\delta_z)^2}r_e^{(t_{now})} \leq r_e^{(t_0)} \leq \frac{(1+\delta_z)^2}{(1-\delta_z)^2}r_e^{(t_{now})}$$

□

As a result, given $\varepsilon_r \leq 1$, if $\delta_z \leq \frac{1}{8}\varepsilon_r$, then $\frac{(1-\delta_z)^2}{(1+\delta_z)^2} \leq (1-8\delta_z)$ and $\frac{(1+\delta_z)^2}{(1-\delta_z)^2} \leq (1+8\delta_z)$

$$(1-8\delta_z) \leq r_e^{(t_0)} \leq (1+8\delta_z)r_e^{(t_{now})}$$

$$(1-\varepsilon_r) \leq r_e^{(t_0)} \leq (1+\varepsilon_r)r_e^{(t_{now})}.$$

Analysis of P

Here we remind that in our system,

$$c_e^{(t)} = \frac{1}{r_e^{(t)}}$$
$$P_{e=uv}^{(t)} = \frac{c_e^{(t)}}{\sum_{v' \in N(u)} c_{uv'}^{(t)}}$$

$P^{(t)}$ serves as the transition matrix of time t.

Lemma 4.4.8. *For our system, assume for $\varepsilon_r \geq 0$, there are T_{intl}, T_{conv}, such that $t_{now} \geq T_{intl} + T_{conv}$, $\forall t_0$ that $t_{now} - T_{conv} \leq t_0 \leq t_{now}$ we know $(1-\varepsilon_r) r_e^{(t_{now})} \leq r_e^{(t_0)} \leq (1+\varepsilon_r) r_e^{(t_{now})}$, then for $t_{now} \geq T_{intl} + T_{conv}$, $\forall t_0$ that $t_{now} - T_{conv} \leq t_0 \leq t_{now}$*

$$\frac{(1-\varepsilon_r)}{(1+\varepsilon_r)} P_e^{(t_{now})} \leq P_e^{(t_0)} \leq \frac{(1+\varepsilon_r)}{(1-\varepsilon_r)} P_e^{(t_{now})}, \forall e \in E.$$

Proof. By definition $c_e^{(t)} = \frac{1}{r_e^{(t)}}$, for $t_{now} \geq T_{intl} + T_{conv}$, $\forall t_0$ that $t_{now} - T_{conv} \leq t_0 \leq t_{now}$ from the assumption $(1-\varepsilon_r) r_e^{(t_{now})} \leq r_e^{(t_0)} \leq (1+\varepsilon_r) r_e^{(t_{now})}$, we know

$$\frac{1}{(1+\varepsilon_r)} c_e^{(t_{now})} \leq c_e^{(t_0)} \leq \frac{1}{(1-\varepsilon_r)} c_e^{(t_{now})}$$

For any $e = uv \in E$, $P_{uv}^{(t_0)} = \frac{c_{uv}^{(t_0)}}{\sum_{v' \in N(u)} c_{uv'}^{(t_0)}}$

$$\frac{(1-\varepsilon_r) c_{uv}^{(t_{now})}}{(1+\varepsilon_r) \sum_{v' \in N(u)} c_{uv'}^{(t_0)}} \leq P_{uv}^{(t_0)} \leq \frac{(1+\varepsilon_r) c_{uv}^{(t_{now})}}{(1-\varepsilon_r) \sum_{v' \in N(u)} c_{uv'}^{(t_0)}}$$

which means

$$\frac{(1-\varepsilon_r)}{(1+\varepsilon_r)} P_e^{(t_{now})} \leq P_e^{(t_0)} \leq \frac{(1+\varepsilon_r)}{(1-\varepsilon_r)} P_e^{(t_{now})}$$

□

In the proof above, given $\delta_P \leq 1$, if $\varepsilon_r \leq \frac{1}{3}\delta_P$, then

$$(1-\delta_P) P_e^{(t_{now})} \leq P_e^{(t_0)} \leq (1+\delta_P) P_e^{(t_{now})}$$

which means

$$\left\|P_e^{(t_0)} - P_e^{(t_{now})}\right\|_\infty \leq \delta_P P_e^{(t_{now})}$$

Analysis of y

$\mathbf{y}^{(t)}$ counts the number of fractional tokens on every node at time t.

Lemma 4.4.9. *Let $\boldsymbol{y}$ be a non-negative row vector, $P \in \mathbb{R}^{n\times n}$ a matrix s.t. $\forall k, \sum_{j=1}^{n} P_{kj} \leq 1$, then $\|\boldsymbol{y}\cdot P\|_1 \leq \|\boldsymbol{y}\|_1$.*

Proof.

$$\begin{aligned}\|\mathbf{y}\cdot P\|_1 &= \sum_{j=1}^{n}\left(\sum_{k=1}^{n} y_k P_{kj}\right)\\ &= \sum_{k=1}^{n}\left(y_k \sum_{j=1}^{n} P_{kj}\right)\\ &\leq \sum_{k=1}^{n} y_k\end{aligned}$$

□

We remind that $\underline{P}$ for a matrix obtained by putting the last row and column of P by zeros. We now consider the linear system: $\mathbf{y}^{(t+1)} = \mathbf{y}^{(t)}\underline{P}^{(t)} + \underline{\mathbf{b}}$

Lemma 4.4.10. *For given ε_y , let $T_y = 2n\Delta R_{ratio}\ln(\frac{2\sqrt{n\Delta R_{ratio}}}{\varepsilon_y})$, assume we have T_{intl} and T_{conv} that $T_{conv} \geq T_y$, such that $t \geq T_{intl} + T_{conv}$, $\forall k$ that $t - T_{conv} \leq k \leq t$, $\left|\left(\underline{P}^{(k)} - \underline{P}^{(t)}\right)\right| \leq \delta_P \cdot \underline{P}^{(t)}$ for some $\delta_P \leq \frac{\varepsilon_y}{2(T_{conv})^2}$, then for $t \geq T_{intl} + T_{conv}$, $\forall t_0$ that $t - T_{conv} \leq t_0 \leq t$*

$$\left\|\boldsymbol{y}^{(t)} - \underline{\boldsymbol{b}}\sum_{k=0}^{t-t_0-1}\left(\underline{P}^{(t)}\right)^k\right\|_1 \leq \varepsilon_y \left\|\boldsymbol{y}^{(t)}\right\|_1$$

Proof. Define

$$\Phi(l,k) = \begin{cases}\underline{P}^{(l)}\underline{P}^{(l+1)}\cdots\underline{P}^{(k-1)} & 0 < l < k\\ I & k = l\end{cases} \tag{4.29}$$

then

$$\mathbf{y}^{(t)} = \mathbf{y}^{(t_0)}\Phi(t_0,t) + \underline{\mathbf{b}}\sum_{k=t_0+1}^{t}\Phi(k,t) \tag{4.30}$$

For $t = t_0 + T$, we regard $\Phi(t_0, t)$,

$$\begin{aligned}
\Phi(t_0, t) &= \Phi(t_0, t-1)\underline{P}^{(t-1)} \\
&= \Phi(t_0, t-1)\underline{P}^{(t)} + \Phi(t_0, t-1)\left(\underline{P}^{(t-1)} - \underline{P}^{(t)}\right) \\
&= \Phi(t_0, t_0)\left(\underline{P}^{(t)}\right)^{t-t_0} + \sum_{k=t_0}^{t-1} \Phi(t_0, k) \cdot \left(\underline{P}^{(k)} - \underline{P}^{(t)}\right)\left(\underline{P}^{(t)}\right)^{t-k-1} \\
&= \left(\underline{P}^{(t)}\right)^{t-t_0} + \sum_{k=t_0}^{t-1} \Phi(t_0, k) \cdot \left(\underline{P}^{(k)} - \underline{P}^{(t)}\right)\left(\underline{P}^{(t)}\right)^{t-k-1}
\end{aligned}$$

Then

$$\mathbf{y}^{(t_0)}\Phi(t_0, t) = \mathbf{y}^{(t_0)}\left(\underline{P}^{(t)}\right)^{t-t_0} + \mathbf{y}^{(t_0)}\sum_{k=t_0}^{t-1} \Phi(t_0, k) \cdot \left(\underline{P}^{(k)} - \underline{P}^{(t)}\right)\left(\underline{P}^{(t)}\right)^{t-k-1} \quad (4.31)$$

(1) We first analyze $\left(\underline{P}^{(t)}\right)^{t-t_0}$.

Because $P^{(t)}$ is a stochastic matrix and $\underline{P}^{(t)}$ obtained by putting the last row and column of P into zeros, then by (Lemma 4.3 of) [6] , the spectral radius $\rho(\underline{P}^{(t)})$ (the largest absolute value of its eigenvalues) of $\underline{P}^{(t)}$ has $\rho(\underline{P}^{(t)}) < 1$. We denote $\mathrm{vol}_u^{(t)} = \sum_{v \in N(u)} c_{uv}^{(t)}$, and $\mathrm{vol}_{\max}^{(t)}$ $\mathrm{vol}_{\min}^{(t)}$ the maximal/minimal $\mathrm{vol}_u^{(t)}$ in $u \in V$.

By the calculation in Section 3.1 of [6] , $\left\|\left(\underline{P}^{(t)}\right)^{t-t_0}\right\|_2$ could be bounded by:

$$\left\|\left(\underline{P}^{(t)}\right)^{t-t_0}\right\|_2 \leq \sqrt{\frac{\mathrm{vol}_{\max}^{(t)}}{\mathrm{vol}_{\min}^{(t)}}}\left(\rho(\underline{P}^{(t)})\right)^{t-t_0}$$

and to upper bound $\rho(\underline{P}^{(t)})$, the Theorem 4.9 and Theorem 4.11 of [6]) shows

$$\begin{aligned}
\rho(\underline{P}^{(t)}) &\leq \left(1 - \frac{\lambda_2(\underline{P}^{(t)})}{2n\mathrm{vol}_{\max}^{(t)}} \sum_{u \in V, u \neq sink} \frac{c_{u,sink}^{(t)}}{c_{u,sink}^{(t)} + \lambda_2(\underline{P}^{(t)})}\right) \\
&\leq \left(1 - \frac{\lambda_2(\underline{P}^{(t)})}{2n\mathrm{vol}_{\max}^{(t)}} \frac{c_{min}^{(t)}}{\lambda_2(\underline{P}^{(t)})}\right) \\
&\leq \left(1 - \frac{c_{min}^{(t)}}{2n\mathrm{vol}_{\max}^{(t)}}\right)
\end{aligned} \quad (4.32)$$

We define

$$r_{min}^{(t)} = \min_{e \in E} r_e^{(t)}, \qquad r_{max}^{(t)} = \max_{e \in E} r_e^{(t)}$$

$$R_{min} = r_{min}^{(t)}, \forall t, \qquad R_{max} = r_{max}^{(t)}, \forall t$$

and $c_{min}^{(t)}, c_{max}^{(t)}, C_{min}, C_{max}$ respectively.

Let R_{ratio} denote $R_{ratio} = \frac{r_{max}^{(t)}}{r_{min}^{(t)}} = \frac{c_{max}^{(t)}}{c_{min}^{(t)}}, \forall t$, we could have the following inequalities:

$$\begin{aligned}
\mathrm{vol}_{\max}^{(t)} &\leq \Delta c_{max}^{(t)} \\
\mathrm{vol}_{\min}^{(t)} &\geq c_{min}^{(t)} \\
\frac{c_{min}^{(t)}}{\mathrm{vol}_{\max}^{(t)}} &\geq \frac{1}{\Delta R_{ratio}} \\
\frac{\mathrm{vol}_{\max}^{(t)}}{\mathrm{vol}_{\min}^{(t)}} &\leq \Delta R_{ratio}
\end{aligned}$$

In fact we could bound R_{ratio}. For $t \geq T_{intl}$, $r_{max}^{(t)} \leq \frac{6}{5} \left\| \boldsymbol{w}^{(t)} \right\|_1$ and $r_{max}^{(t)} \leq \frac{\varepsilon}{5mU_{max}^2} \left\| \boldsymbol{w}^{(t)} \right\|_1$ so $R_{ratio} = \frac{r_{max}^{(t)}}{r_{min}^{(t)}} \leq 6U_{max}^2 \frac{m}{\varepsilon}$ and $c_{max}^{(t)} \leq U_{max}^2$. But let's take the notation R_{ratio} to simplify the calculation.

$$\rho(\underline{P}^{(t)}) \leq \left(1 - \frac{1}{2n\Delta R_{ratio}}\right)$$

$$\begin{aligned}
\left\| \left(\underline{P}^{(t)}\right)^{t-t_0} \right\|_2 &\leq \sqrt{\Delta R_{ratio}} \left(\rho(\underline{P}^{(t)})\right)^{t-t_0} \\
&\leq \sqrt{\Delta R_{ratio}} \left(1 - \frac{1}{2n\Delta R_{ratio}}\right)^{t-t_0}
\end{aligned}$$

And $\left\| \mathbf{y}^{(t_0)} \right\|_2 \leq \left\| \mathbf{y}^{(t_0)} \right\|_1 \leq \left\| \mathbf{y}^{(t)} \right\|_1$,

$$\left\| \mathbf{y}^{(t_0)} \left(\underline{P}^{(t)}\right)^{t-t_0} \right\|_2 \leq \sqrt{\Delta R_{ratio}} \left(1 - \frac{1}{2n\Delta R_{ratio}}\right)^{t-t_0} \left\| \mathbf{y}^{(t)} \right\|_1$$

given δ_y, let

$$T_y = 2n\Delta R_{ratio} \ln(\frac{\sqrt{n\Delta R_{ratio}}}{\delta_y}) \geq \frac{\ln(\frac{\delta_y}{\sqrt{n\Delta R_{ratio}}})}{\ln(1 - \frac{1}{2n\Delta R_{ratio}})}$$

then if $t - t_0 \geq T_y$ then

$$\left\|\mathbf{y}^{(t_0)}\left(\underline{P}^{(t)}\right)^{t-t_0}\right\|_2 \leq \frac{\delta_y}{\sqrt{n}}\left\|\mathbf{y}^{(t)}\right\|_1$$

$$\left\|\mathbf{y}^{(t_0)}\left(\underline{P}^{(t)}\right)^{t-t_0}\right\|_1 \leq \delta_y\left\|\mathbf{y}^{(t)}\right\|_1$$

Let $\delta_y \leq \frac{\varepsilon_y}{2}$
then if $t - t_0 \geq T_y$ then

$$\left\|\mathbf{y}^{(t_0)}\left(\underline{P}^{(t)}\right)^{t-t_0}\right\|_1 \leq \frac{\varepsilon_y}{2}\left\|\mathbf{y}^{(t)}\right\|_1$$

(2) Then we analyze $\mathbf{y}^{(t_0)}\Phi(t_0,t) - \mathbf{y}^{(t_0)}\left(\underline{P}^{(t)}\right)^{t-t_0}$ and $\underline{\mathbf{b}}\sum_{k=t_0+1}^{t}\Phi(k,t)$

$$\begin{aligned}\left\|\mathbf{y}^{(t_0)}\Phi(t_0,t) - \mathbf{y}^{(t_0)}\left(\underline{P}^{(t)}\right)^{t-t_0}\right\|_1 &= \left\|\mathbf{y}^{(t_0)}\sum_{k=t_0}^{t-1}\Phi(t_0,k)\cdot\left(\underline{P}^{(k)} - \underline{P}^{(t)}\right)\left(\underline{P}^{(t)}\right)^{t-k-1}\right\|_1 \\ &= \left\|\sum_{k=t_0}^{t-1}\mathbf{y}^{(t_0)}\Phi(t_0,k)\cdot\left(\underline{P}^{(k)} - \underline{P}^{(t)}\right)\left(\underline{P}^{(t)}\right)^{t-k-1}\right\|_1\end{aligned}$$

From the assumption $\forall t_0 \leq k \leq t, \left|\left(\underline{P}^{(k)} - \underline{P}^{(t)}\right)\right| \leq \delta_P \cdot \underline{P}^{(t)}$, so the sum of every row of $\left(\frac{1}{\delta_P}\left|\underline{P}^{(k)} - \underline{P}^{(t)}\right|\right)$ is less than 1.

And we know also $\underline{P}^{(t_0)}$, $\Phi(t_0,k)$ and $\left(\underline{P}^{(t)}\right)^{t-k-1}$ are matrices such that the sum of every row is less than 1, then from Lemma 4.4.9 $\left\|\mathbf{y}^{(t_0)}\Phi(t_0,k)\right\|_1 \leq \left\|\mathbf{y}^{(t_0)}\right\|_1$, and we have

$$\begin{aligned}\left\|\mathbf{y}^{(t_0)}\Phi(t_0,k)\cdot\left(\underline{P}^{(k)} - \underline{P}^{(t)}\right)\right\|_1 &\leq \left\|\delta_P\cdot\mathbf{y}^{(t_0)}\Phi(t_0,k)\cdot\left(\frac{1}{\delta_P}\left|\underline{P}^{(k)} - \underline{P}^{(t)}\right|\right)^{(t)}\right\|_1 \\ &\leq \delta_P\left\|\mathbf{y}^{(t_0)}\right\|_1\end{aligned}$$

$$\left\|\sum_{k=t_0}^{t-1}\mathbf{y}^{(t_0)}\Phi(t_0,k)\cdot\left(\underline{P}^{(k)} - \underline{P}^{(t)}\right)\left(\underline{P}^{(t)}\right)^{t-k-1}\right\|_1 \leq \delta_P(t-t_0)\left\|\mathbf{y}^{(t_0)}\right\|_1$$

Similarly, we do the same analysis for $\underline{\mathbf{b}} \cdot \Phi(k,t)$, $\forall t_0 \leq k \leq t$,

$$\begin{aligned}
\Phi(k,t) &= \Phi(k,t-1)\underline{P}^{(t-1)} \\
&= \Phi(k,t-1)\underline{P}^{(t)} + \Phi(k,t-1)(\underline{P}^{(t-1)} - \underline{P}^{(t)}) \\
&= \left(\underline{P}^{(t)}\right)^{t-k} + \sum_{j=k}^{t-1} \Phi(k,j) \cdot \left(\underline{P}^{(j)} - \underline{P}^{(t)}\right)\left(\underline{P}^{(t)}\right)^{t-j-1}
\end{aligned}$$

Then

$$\begin{aligned}
\left\| \underline{\mathbf{b}} \cdot \Phi(k,t) - \underline{\mathbf{b}} \left(\underline{P}^{(t)}\right)^{t-k} \right\|_1 &\leq \delta_P (t-t_0) \left\|\underline{\mathbf{b}}\right\|_1 \\
\left\| \sum_{k=t_0+1}^{t} \underline{\mathbf{b}} \cdot \Phi(k,t) - \sum_{k=t_0+1}^{t} \underline{\mathbf{b}} \left(\underline{P}^{(t)}\right)^{t-k} \right\|_1 &\leq \delta_P (t-k)(t-t_0) \left\|\underline{\mathbf{b}}\right\|_1 \\
&\leq \delta_P (t-t_0)^2 \left\|\underline{\mathbf{b}}\right\|_1
\end{aligned}$$

We know by definition that $\left\|\underline{\mathbf{b}}\right\|_1 \leq \left\|\mathbf{y}^{(t)}\right\|_1$ then for $\mathbf{y}^{(t)} = \mathbf{y}^{(t_0)}\Phi(t_0,t) + \underline{\mathbf{b}} \sum_{k=t_0+1}^{t} \Phi(k,t)$, we get

$$\begin{aligned}
\left\| \mathbf{y}^{(t)} - \mathbf{y}^{(t_0)} \left(\underline{P}^{(t)}\right)^{t-t_0} - \underline{\mathbf{b}} \sum_{k=t_0+1}^{t} \left(\underline{P}^{(t)}\right)^{t-k} \right\|_1 &\leq \delta_P (t-t_0) \left\|\mathbf{y}^{(t)}\right\|_1 + \delta_P (t-t_0)^2 \left\|\underline{\mathbf{b}}\right\|_1 \\
&\leq 2\delta_P (t-t_0)^2 \left\|\mathbf{y}^{(t)}\right\|_1 \\
&= 2\delta_P (T_y)^2 \left\|\mathbf{y}^{(t)}\right\|_1
\end{aligned}$$

if

$$\delta_P \leq \frac{\varepsilon_y}{2(T_{conv})^2}$$

for $t - t_0 = T_y$

$$\left\| \mathbf{y}^{(t)} - \mathbf{y}^{(t_0)} \left(\underline{P}^{(t)}\right)^{t-t_0} - \underline{\mathbf{b}} \sum_{k=t_0+1}^{t} \left(\underline{P}^{(t)}\right)^{t-k} \right\|_1 \leq \frac{\varepsilon_y}{2} \left\|\mathbf{y}^{(t)}\right\|_1$$

and

$$\left\| \mathbf{y}^{(t_0)} \left(\underline{P}^{(t)}\right)^{t-t_0} \right\|_1 \leq \frac{\varepsilon_y}{2} \left\|\mathbf{y}^{(t)}\right\|_1$$

$$\left\|\mathbf{y}^{(t)}-\underline{\mathbf{b}}\sum_{k=t_0+1}^{t}\left(\underline{P}^{(t)}\right)^{t-k}\right\|_1 \leq \left\|\mathbf{y}^{(t)}-\mathbf{y}^{(t_0)}\left(\underline{P}^{(t)}\right)^{t-t_0}-\underline{\mathbf{b}}\sum_{k=t_0+1}^{t}\left(\underline{P}^{(t)}\right)^{t-k}\right\|_1+\left\|\mathbf{y}^{(t_0)}\left(\underline{P}^{(t)}\right)^{t-t_0}\right\|_1$$
$$\leq \frac{\varepsilon_y}{2}\left\|\mathbf{y}^{(t)}\right\|_1+\frac{\varepsilon_y}{2}\left\|\mathbf{y}^{(t)}\right\|_1$$
$$\leq \varepsilon_y\left\|\mathbf{y}^{(t)}\right\|_1$$

Equivalently,

$$\left\|\mathbf{y}^{(t)}-\underline{\mathbf{b}}\sum_{k=0}^{t-t_0-1}\left(\underline{P}^{(t)}\right)^{k}\right\|_1 \leq \varepsilon_y\left\|\mathbf{y}^{(t)}\right\|_1$$

□

Analysis of $\boldsymbol{\phi}$

We are going to show that $\boldsymbol{\phi}^{(t)}=\mathbf{y}^{(t)}\left(D^{(t)}\right)^{-1}$ is approximating the potentials.

Lemma 4.4.11. *For given δ_ϕ , let $T_h \geq 2n\Delta R_{ratio}\ln\frac{4n\Delta^2 R_{ratio}h_{max}^{(\infty)}\sqrt{n\Delta R_{ratio}}}{\delta_\phi}$, assume we have T_{intl} and T_{conv} that $T_{conv}\geq T_h$, such that $t\geq T_{intl}+T_{conv}$, $\forall t_0$ that $t-T_{conv}\leq t_0\leq t-T_h$, $\left\|\boldsymbol{y}^{(t)}-\underline{\boldsymbol{b}}\sum_{k=0}^{t-t_0-1}\left(\underline{P}^{(t)}\right)^{k}\right\|_1\leq\varepsilon_y\left\|\boldsymbol{y}^{(t)}\right\|_1$, for $\varepsilon_y=\frac{\delta_\phi}{4\Delta R_{ratio}\cdot nh_{max}^{(\infty)}}$ then for $t_{now}\geq T_{intl}+T_{conv}$, let $\boldsymbol{\phi}^{(t_{now})}=\boldsymbol{y}^{(t_{now})}\left(D^{(t_{now})}\right)^{-1}$, we have*

$$\left\|\boldsymbol{\phi}^{(t_{now})}-\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})},D^{(t_{now})})\right\|_1\leq\delta_\phi$$

where $\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})},D^{(t_{now})})$ is the potential of electrical network with resistances $\boldsymbol{r}^{(t)}$ and current flow value F.

Proof. For $t_{now}\geq T_{intl}+T_{conv}$, $t_{now}-T_{conv}\leq t_0\leq t_{now}-T_h$, by assumption $\left\|\mathbf{y}^{(t_{now})}-\underline{\mathbf{b}}\sum_{k=0}^{t_{now}-t_0-1}\left(\underline{P}^{(t_{now})}\right)^{k}\right\|_1\leq\varepsilon_y\left\|\mathbf{y}^{(t_{now})}\right\|_1$.

For all $t\geq 0$, consider the electrical network with resistors $\mathbf{r}^{(t)}$ and constant current source of current value F, we denote $c_{uv}^{(t)}=\frac{1}{r_{uv}^{(t)}}$ are conductances.

We also denote $A_{u,v}^{(t)}=c_{e=uv}^{(t)}$, $D_{u,u}^{(t)}=\sum_{v'\in N(u)}c_{uv'}^{(t)}$, $D_{u,v}^{(t)}=0, u\neq v$, $L^{(t)}=D^{(t)}-A^{(t)}$, $\mathrm{vol}^{(t)}(u)=D_{u,u}^{(t)}$, $\underline{P}^{(t)}$ is the same as our system and in fact, $\underline{P}^{(t)}=\left(D^{(t)}\right)^{-1}A^{(t)}$ and $P^{(t)}$ is a stochastic matrix. $\underline{P}^{(t)}$ is obtained by putting the last row and column of P into zeros, and $\underline{\rho}^{(t)}$ is the spectral radius of $\underline{P}^{(t)}$.

By basic algebras we know that a vector ϕ is potential vector induced by current source of value 1 and $\mathbf{r}^{(t)}$ if $L^{(t)}\phi^\top=\mathbf{b}^\top$.

For any $n \times n$ matrices A, B such that $\lim_{i\to\infty} A^i = 0$ and B invertible, given time i we denote $\boldsymbol{h}^{(i)}(A,B) = \underline{\mathbf{b}}\left(\sum_{k=0}^{i}(A)^k\right)(B)^{-1}$, and $\boldsymbol{h}^{(\infty)}(A,B) = \underline{\mathbf{b}}\left(\sum_{k=0}^{\infty}(A)^k\right)(B)^{-1}$ note that since assumed $\lim_{i\to\infty} A^i = 0$, the later sum exists and converges to $(I-A)^{-1}B^{-1}$.

From [6] we know that

$$\begin{aligned}\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) &= \underline{\mathbf{b}}\left(\sum_{k=0}^{\infty}\left(\underline{P}^{(t_{now})}\right)^k\right)\left(D^{(t_{now})}\right)^{-1} \\ &= \underline{\mathbf{b}}\left(I-\underline{P}^{(t_{now})}\right)^{-1}\left(D^{(t_{now})}\right)^{-1}\end{aligned}$$

(1) We first show that $\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})})$ is the the potential vector induced by $\mathbf{r}^{(t_{now})}$, namely $L^{(t_{now})}\left(\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})})\right)^{\top} = \mathbf{b}^{\top}$.

Note that $D^{(t_{now})} - \underline{A}^{(t_{now})}$ is a symmetric matrix, then

$$\begin{aligned}L^{(t_{now})}\left(\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})})\right)^{\top} &= L^{(t_{now})}\left(\underline{\mathbf{b}}\left(I-\underline{P}^{(t_{now})}\right)^{-1}\left(D^{(t_{now})}\right)^{-1}\right)^{\top} \\ &= L^{(t_{now})}\left(\underline{\mathbf{b}}\left(D^{(t_{now})}-\underline{A}^{(t_{now})}\right)^{-1}\right)^{\top} \\ &= L^{(t_{now})}\left(\left(D^{(t_{now})}-\underline{A}^{(t_{now})}\right)^{-1}\right)^{\top}\underline{\mathbf{b}}^{\top} \\ &= L^{(t_{now})}\left(D^{(t_{now})}-\underline{A}^{(t_{now})}\right)^{-1}\underline{\mathbf{b}}^{\top} \\ &= L^{(t_{now})}\left(I-\underline{P}^{(t_{now})}\right)^{-1}\left(D^{(t_{now})}\right)^{-1}\underline{\mathbf{b}}^{\top} \\ &= \mathbf{b}^{\top}\end{aligned}$$

Which means $\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})})$ is the exact potential induced by $\mathbf{r}^{(t_{now})}$. We denote $h_{max}^{(\infty)}$ for $\left\|\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})})\right\|_{\infty}$ in the following, and fact this value corresponds to the potential of source node.

(2) Then we regard $\boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) = \underline{\mathbf{b}}\left(\sum_{k=0}^{T_h}\left(\underline{P}^{(t_{now})}\right)^k\right)\left(D^{(t_{now})}\right)^{-1}$, and bound the difference between $\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})})$ and $\boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})}, D^{(t_{now})})$:

$$
\begin{aligned}
&\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) - \boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) \\
&= \underline{\mathbf{b}} \left(\sum_{k=0}^{\infty} \left(\underline{P}^{(t_{now})} \right)^k - \sum_{k=0}^{T_h} \left(\underline{P}^{(t_{now})} \right)^k \right) \left(D^{(t_{now})} \right)^{-1} \\
&= \underline{\mathbf{b}} \left(\sum_{k=T_h+1}^{\infty} \left(\underline{P}^{(t_{now})} \right)^k \right) \left(D^{(t_{now})} \right)^{-1}
\end{aligned}
$$

From [6] the proof of theorem 4.4, we know

$$
\left\| \underline{\mathbf{b}} \left(\sum_{k=T_h+1}^{\infty} \left(\underline{P}^{(t_{now})} \right)^k \right) \left(D^{(t_{now})} \right)^{-1} \right\|_2 \leq F \cdot \sqrt{\frac{\mathrm{vol}_{max}(\underline{P}^{(t_{now})})}{\mathrm{vol}_{min}(\underline{P}^{(t_{now})})}} \frac{\left(\rho(\underline{P}^{(t_{now})}) \right)^{T_h}}{(1 - \rho(\underline{P}^{(t_{now})}))} \frac{1}{\mathrm{vol}_{source}(\underline{P}^{(t_{now})}))}
$$

where $\rho(\underline{P}^{(t_{now})})$ is the spectral radius (the largest absolute value of its eigenvalues) of $\underline{P}^{(t_{now})}$.

Because $\frac{\mathrm{vol}_{max}(\underline{P}^{(t_{now})})}{\mathrm{vol}_{min}(\underline{P}^{(t_{now})})} \leq \Delta R_{ratio}$ and

$$
\frac{F}{\mathrm{vol}_{source}(\underline{P}^{(t_{now})}))} = \sum_{(source,v) \in E} F \cdot r_{source,v}^{(t_{now})} \leq \Delta h_{max}^{(\infty)}
$$

we know

$$
\begin{aligned}
&\left\| \boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) - \boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) \right\|_2 \\
&= \left\| \underline{\mathbf{b}} \left(\sum_{k=T_h+1}^{\infty} \left(\underline{P}^{(t_{now})} \right)^k \right) \left(D^{(t_{now})} \right)^{-1} \right\|_2 \\
&\leq F \cdot \sqrt{\Delta R_{ratio}} \frac{\left(\rho(\underline{P}^{(t_{now})}) \right)^{T_h}}{(1 - \rho(\underline{P}^{(t_{now})}))} \frac{1}{\mathrm{vol}_{source}(\underline{P}^{(t_{now})}))} \\
&\leq \sqrt{\Delta R_{ratio}} \frac{\left(\rho(\underline{P}^{(t_{now})}) \right)^{T_h}}{(1 - \rho(\underline{P}^{(t_{now})}))} \Delta h_{max}^{(\infty)}
\end{aligned}
$$

From Formula 4.32 we know

$$
\begin{aligned}
\rho(\underline{P}^{(t)}) &\leq \left(1 - \frac{1}{2n \mathrm{vol}_{\max}^{(t)}} \right) \\
&\leq \left(1 - \frac{1}{2n \Delta R_{ratio}} \right)
\end{aligned}
$$

Thus if $T_h \geq 2n\Delta R_{ratio} \ln \frac{2n\Delta^2 R_{ratio} h_{max}^{(\infty)} \sqrt{n\Delta R_{ratio}}}{\delta_h}$ we have

$$\left\| \boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) - \boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) \right\|_2 \leq \frac{\delta_h}{\sqrt{n}}$$
$$\left\| \boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) - \boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) \right\|_1 \leq \delta_h$$

Let $\delta_h = \frac{\delta_\phi}{2}$,

$$\left\| \boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) - \boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) \right\|_1 \leq \frac{\delta_\phi}{2}$$

(3) We then consider $\boldsymbol{\phi}^{(t_{now})} = \mathbf{y}^{(t_{now})} \left(D^{(t_{now})}\right)^{-1}$.

Since $\left\| \mathbf{y}^{(t_{now})} - \underline{\mathbf{b}} \sum_{k=0}^{T_h} \left(\underline{P}^{(t_{now})}\right)^k \right\|_1 \leq \varepsilon_y \left\| \mathbf{y}^{(t_{now})} \right\|_1$, we know $(1-\varepsilon_y) \left\| \mathbf{y}^{(t_{now})} \right\|_1 \leq \left\| \underline{\mathbf{b}} \sum_{k=0}^{T_h} \left(\underline{P}^{(t_{now})}\right)^k \right\|_1$, $\left\| \mathbf{y}^{(t_{now})} \right\|_1 \leq \frac{1}{(1-\varepsilon_y)} \left\| \underline{\mathbf{b}} \sum_{k=0}^{T_h} \left(\underline{P}^{(t_{now})}\right)^k \right\|_1$.

And $\underline{\mathbf{b}} \sum_{k=0}^{T_h} \left(\underline{P}^{(t_{now})}\right)^k = \boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})} D^{(t_{now})}) \cdot D^{(t_{now})}$.

For $\boldsymbol{\phi}^{(t_{now})} = \mathbf{y}^{(t_{now})} \left(D^{(t_{now})}\right)^{-1}$,

$$\begin{aligned}
\left\| \boldsymbol{\phi}^{(t_{now})} - \boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) \right\|_1 &= \left\| \mathbf{y}^{(t_{now})} \left(D^{(t_{now})}\right)^{-1} - \sum_{k=0}^{T_h} \left(\underline{P}^{(t_{now})}\right)^k \left(D^{(t_{now})}\right)^{-1} \right\|_1 \\
&= \left\| \left(\mathbf{y}^{(t_{now})} - \underline{\mathbf{b}} \sum_{k=0}^{T_h} \left(\underline{P}^{(t_{now})}\right)^k \right) \left(D^{(t_{now})}\right)^{-1} \right\|_1 \\
&\leq R_{max} \cdot \left\| \left(\mathbf{y}^{(t_{now})} - \underline{\mathbf{b}} \sum_{k=0}^{T_h} \left(\underline{P}^{(t_{now})}\right)^k \right) \right\|_1 \\
&\leq R_{max} \cdot \varepsilon_y \left\| \mathbf{y}^{(t_{now})} \right\|_1 \\
&\leq \frac{\varepsilon_y}{(1-\varepsilon_y)} R_{max} \left\| \underline{\mathbf{b}} \sum_{k=0}^{T_h} \left(\underline{P}^{(t_{now})}\right)^k \right\|_1
\end{aligned} \tag{4.33}$$

$$\begin{aligned}
\left\| \underline{\mathbf{b}} \sum_{k=0}^{T_h} \left(\underline{P}^{(t_{now})}\right)^k \right\|_1 &\leq \left\| \underline{\mathbf{b}} \sum_{k=0}^{\infty} \left(\underline{P}^{(t_{now})}\right)^k \right\|_1 \\
&= \left\| \boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) D^{(t_{now})} \right\|_1 \\
&\leq \Delta C_{max} \left\| \boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) \right\|_1
\end{aligned}$$

Note that $\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})})$ are potentials induced by F and $\mathbf{r}^{(t_{now})}$,

$$\begin{aligned}\left\|\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})})\right\|_1 &\leq n\left\|\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})})\right\|_\infty \\ &\leq nh_{max}^{(\infty)}\end{aligned}$$

$$\left\|\underline{\mathbf{b}}\sum_{k=0}^{T_h}\left(\underline{P}^{(t_{now})}\right)^k\right\|_1 \leq \Delta C_{max}\cdot nh_{max}^{(\infty)}$$

From Formula 4.33, when $\varepsilon_y \leq 1/2$

$$\begin{aligned}\left\|\boldsymbol{\phi}^{(t_{now})} - \boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})}, D^{(t_{now})})\right\|_1 &\leq \frac{\varepsilon_y}{(1-\varepsilon_y)}R_{max}\left\|\underline{\mathbf{b}}\sum_{k=0}^{T_h}\left(\underline{P}^{(t_{now})}\right)^k\right\|_1 \\ &\leq \frac{\varepsilon_y}{(1-\varepsilon_y)}\Delta R_{ratio}\cdot nh_{max}^{(\infty)} \\ &\leq 2\varepsilon_y\Delta R_{ratio}\cdot nh_{max}^{(\infty)}\end{aligned}$$

For $\varepsilon_y = \frac{\delta_\phi}{4\Delta R_{ratio}\cdot nh_{max}^{(\infty)}}$,

$$\left\|\boldsymbol{\phi}^{(t_{now})} - \boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})}, D^{(t_{now})})\right\|_1 \leq \frac{\delta_\phi}{2}$$

Then

$$\begin{aligned}&\left\|\boldsymbol{\phi}^{(t_{now})} - \boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})})\right\|_1 \leq \left\|\boldsymbol{\phi}^{(t_{now})} - \boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})}, D^{(t_{now})})\right\|_1 + \\ &\left\|\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})}) - \boldsymbol{h}^{(T_h)}(\underline{P}^{(t_{now})}, D^{(t_{now})})\right\|_1 \\ &\leq \frac{\delta_\phi}{2} + \frac{\delta_\phi}{2} \\ &\leq \delta_\phi\end{aligned}$$

□

Because $\boldsymbol{h}^{(\infty)}(\underline{P}^{(t_{now})}, D^{(t_{now})})$ are potentials induced by $\mathbf{r}^{(t_{now})}$, $\boldsymbol{\phi}^{(t_{now})}$ is an approximation.

Analysis of $\boldsymbol{f}$

Let $\tilde{\boldsymbol{f}}^{(t)}$ be the exact the electrical flow induced by $\mathbf{r}^{(t)}$ with constant current source of current value F and $\boldsymbol{h}^{(\infty)}$ are potentials associated, $c_{uv}^{(t)} = \frac{1}{r_{uv}^{(t)}}$ are conductances. Given an $s-t$ flow $\boldsymbol{f}$, the energy of $\boldsymbol{f}$ with $\boldsymbol{r}$ is defined by $\mathcal{E}_{\boldsymbol{r}}(\boldsymbol{f}) = \sum_{e\in E} r_e(f_e^2)$.

Lemma 4.4.12. *For any $\varepsilon_{en} \geq 0$, if we can compute a vector of $\boldsymbol{\phi}^{(t)}$, such that $\left\|\boldsymbol{\phi}^{(t)} - \boldsymbol{h}^{(\infty)}\right\|_\infty \leq \delta_\phi$ where $\delta_\phi = \frac{\varepsilon_{en} h_{max}^{(\infty)}}{8mR_{ratio}}$ and let $\boldsymbol{f}^{(t)}$ be defined by $f_{uv}^{(t)} = \left|(\phi_u^{(t)} - \phi_v^{(t)})c_{uv}^{(t)}\right|$, then we have:*

$$\left|\mathcal{E}_{\boldsymbol{r}^{(t)}}(\boldsymbol{f}^{(t)}) - \mathcal{E}_{\boldsymbol{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)})\right| \leq \varepsilon_{en} \cdot \mathcal{E}_{\boldsymbol{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)})$$

Proof. By definition, $\tilde{f}_{uv}^{(t)} = \left|(h_u^{(\infty)} - h_v^{(\infty)})c_{uv}^{(t_0)}\right|$ and $f_{uv}^{(t)} = \left|(\phi_u^{(t)} - \phi_v^{(t)})c_{uv}^{(t)}\right|$, then we look at the energy of $\boldsymbol{f}^{(t)}$ and $\tilde{\boldsymbol{f}}^{(t)}$:

$$\mathcal{E}_{\boldsymbol{r}^{(t)}}(\boldsymbol{f}^{(t)}) = \sum_{uv\in E} r_{uv}^{(t)} \left(f_{uv}^{(t)}\right)^2 = \sum_{uv\in E} (\phi_u^{(t)} - \phi_v^{(t)})^2 c_{uv}^{(t)}$$

$$\mathcal{E}_{\boldsymbol{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)}) = \sum_{uv\in E} r_{uv}^{(t)} \left(\tilde{f}_{uv}^{(t)}\right)^2 = \sum_{uv\in E} (h_u^{(\infty)} - h_v^{(\infty)})^2 c_{uv}^{(t)}$$

For $e = uv \in E$,

$$\begin{aligned}\left|(\phi_u^{(t)} - \phi_v^{(t)})^2 - (h_u^{(\infty)} - h_v^{(\infty)})^2\right| &= \left|(\phi_u^{(t)} - \phi_v^{(t)} + h_u^{(\infty)} - h_v^{(\infty)})(\phi_u^{(t)} - \phi_v^{(t)} - h_u^{(\infty)} + h_v^{(\infty)})\right| \\ &= \left|(\phi_u^{(t)} - \phi_v^{(t)} + h_u^{(\infty)} - h_v^{(\infty)})\right| \cdot \left|(\phi_u^{(t)} - h_u^{(\infty)}) + (h_v^{(\infty)} - \phi_v^{(t)})\right|\end{aligned}$$

we have $\left\|\boldsymbol{\phi}^{(t)} - \boldsymbol{h}^{(\infty)}\right\|_\infty \leq \delta_\phi$ and $h_u^{(\infty)}, h_v^{(\infty)} \geq 0$, then

$$\begin{aligned}\left|(\phi_u^{(t)} - \phi_v^{(t)}) + (h_u^{(\infty)} - h_v^{(\infty)})\right| &\leq \left(\left|\phi_u^{(t)} - \phi_v^{(t)}\right| + \left|h_u^{(\infty)} - h_v^{(\infty)}\right|\right) \\ &\leq \left(2\left|h_u^{(\infty)} - h_v^{(\infty)}\right| + 2\delta_\phi\right)\end{aligned}$$

$$\begin{aligned}\left|(\phi_u^{(t)} - \phi_v^{(t)}) - (h_u^{(\infty)} - h_v^{(\infty)})\right| &= \left|(\phi_u^{(t)} - h_u^{(\infty)}) + (h_v^{(\infty)} - \phi_v^{(t)})\right| \\ &\leq 2\delta_\phi\end{aligned}$$

$$\begin{aligned}\left|(\phi_u^{(t)} - \phi_v^{(t)})^2 - (h_u^{(\infty)} - h_v^{(\infty)})^2\right| &\leq \left(2\left|h_u^{(\infty)} - h_v^{(\infty)}\right| + 2\delta_\phi\right) \cdot 2\delta_\phi \\ &= \left|h_u^{(\infty)} - h_v^{(\infty)}\right| \cdot 4\delta_\phi + 4\delta_\phi^2\end{aligned}$$

$$c_{uv}^{(t)}\left|(\phi_u^{(t)}-\phi_v^{(t)})^2-(h_u^{(\infty)}-h_v^{(\infty)})^2\right| \le c_{uv}^{(t)}\left|h_u^{(\infty)}-h_v^{(\infty)}\right|\cdot 4\delta_\phi + 4\delta_\phi^2 c_{uv}^{(t)}$$
$$= \tilde{f}_{uv}^{(t)}\cdot 4\delta_\phi + 4\delta_\phi^2 c_{uv}^{(t)}$$

Note that $\tilde{f}_{uv}^{(t)} \le F$ because F is the flow value from the source,

$$\left|\mathcal{E}_{\boldsymbol{r}^{(t)}}(\boldsymbol{f}^{(t)}) - \mathcal{E}_{\boldsymbol{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)})\right| = \sum_{uv\in E} c_{uv}^{(t)}\left|(\phi_u^{(t)}-\phi_v^{(t)})^2-(h_u^{(\infty)}-h_v^{(\infty)})^2\right|$$
$$\le \sum_{uv\in E}\left(\tilde{f}_{uv}^{(t)}\cdot 4\delta_\phi + 4\delta_\phi^2 c_{uv}^{(t)}\right)$$
$$\le 4\delta_\phi mF + 4\delta_\phi^2 m c_{uv}^{(t)}$$

Because from property of electrical circuit we know $h_{max}^{(\infty)}$ is exactly the potential of the source and

$$\mathcal{E}_{\boldsymbol{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)}) = Fh_{max}^{(\infty)} \tag{4.34}$$

when $\delta_\phi \le \frac{\varepsilon_{en} h_{max}^{(\infty)}}{8mR_{ratio}}$

$$\left|\mathcal{E}_{\boldsymbol{r}^{(t)}}(\boldsymbol{f}^{(t)}) - \mathcal{E}_{\boldsymbol{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)})\right| \le \frac{\varepsilon_{en}}{2}Fh_{max}^{(\infty)} + \frac{\varepsilon_{en}}{2}\left(h_{max}^{(\infty)}\right)^2\frac{1}{mR_{max}}$$
$$\le \frac{\varepsilon_{en}}{2}Fh_{max}^{(\infty)} + \frac{\varepsilon_{en}}{2}h_{max}^{(\infty)}F$$
$$\le \varepsilon_{en}\cdot\mathcal{E}_{\boldsymbol{r}^{(t)}}(\tilde{\boldsymbol{f}}^{(t)})$$

□

Time Complexity

We now fix the parameters to see the time complexity.

We'd like to make the assumption for 4.4.1 hold, i.e. $\varepsilon_{en} = \varepsilon$, $\delta_\tau = \frac{1}{5}$.

Remember that to make the assumptions of Lemma 4.4.3 to Lemma 4.4.12 to be hold we need $\forall t \ge T_{intl} + T_{conv}$, $t - T_{conv} \le t_0 \le t$ time, we need to have $(1-\delta_w)\boldsymbol{w}^{(t)} \le \boldsymbol{w}^{(t_0)} \le \boldsymbol{w}^{(t)}$.

Note that $t \le T_{intl}$, since $\boldsymbol{w}^{(t)}$ do not change, this holds trivially. For $t \ge T_{intl}$, we know $cong(f_e^{(t)}) \le \rho$ and so

$$w_e^{(t+1)} \le w_e^{(t)}\left(1+\frac{\alpha\varepsilon}{\rho}cong_e(f^{(t)})\right)$$
$$\le w_e^{(t)}(1+\alpha\varepsilon)$$
$$\le w_e^{(t)}\exp(\alpha\varepsilon), \text{ since } 1+x\le e^x \text{ for } x\ge 0$$

For $t \ge T_{intl} + T_{conv}$, $t - T_{conv} \le t_0 \le t$,

$$w_e^{(t)} \leq w_e^{(t_0)} \exp\left((t-t_0)\alpha\varepsilon\right) \leq \exp\left(T_{conv}\cdot\alpha\varepsilon\right)$$

To have $(1-\delta_w)\boldsymbol{w}^{(t)} \leq \boldsymbol{w}^{(t_0)}$, it means $w^{(t)} \leq w^{(t_0)}\frac{1}{(1-\delta_w)}$, and $w^{(t)} \leq w^{(t_0)}(1+\delta_w)$could imply this since $(1+\delta_w) \leq \frac{1}{(1-\delta_w)}$, for $\delta_w \leq 1$.

Thus we need

$$\begin{aligned}\exp\left(T_{conv}\cdot\alpha\varepsilon\right) &\leq (1+\delta_w)\\ \alpha &\leq \frac{\ln(1+\delta_w)}{T_{conv}\cdot\varepsilon}\end{aligned}$$

We recall that $\varepsilon_r \leq \frac{1}{3}\delta_P$, $\delta_z \leq \frac{1}{8}\varepsilon_r$, $\delta_w \leq \frac{1}{5}\delta_z$, then $\delta_w \leq \frac{1}{120}\delta_P$.

$$\alpha \leq \frac{\ln(1+\frac{1}{120}\delta_P)}{T_{conv}\cdot\varepsilon}$$

$\alpha \leq \frac{\delta_P}{240T_{conv}\cdot\varepsilon}$ could imply the above.

We then give a bound of δ_P.
Let $\delta_\phi = \frac{\varepsilon_{en}h_{max}^{(\infty)}}{8mR_{ratio}}$
Then $\delta_h = \frac{\delta_\phi}{2} = \frac{\varepsilon_{en}h_{max}^{(\infty)}}{16mR_{ratio}}$

$$\begin{aligned}\varepsilon_y &= \frac{\delta_\phi}{4\Delta R_{ratio}\cdot nh_{max}^{(\infty)}}\\ &= \frac{1}{4\Delta R_{ratio}\cdot nh_{max}^{(\infty)}}\cdot\frac{\varepsilon_{en}h_{max}^{(\infty)}}{8mR_{ratio}}\\ &= \frac{\varepsilon_{en}}{32\Delta nmR_{ratio}^2}\end{aligned}$$

$\delta_y = \frac{\varepsilon_y}{2} = \frac{\varepsilon_{en}}{64\Delta nmR_{ratio}^2}$

$$\begin{aligned}T_y &= 2n\Delta R_{ratio}\ln(\frac{\sqrt{n\Delta R_{ratio}}}{\delta_y})\\ &= 2n\Delta R_{ratio}\ln(\sqrt{n\Delta R_{ratio}}\frac{64\Delta nmR_{ratio}^2}{\varepsilon_{en}})\\ &= 2n\Delta R_{ratio}\ln(\frac{64\Delta^{\frac{3}{2}}n^{\frac{3}{2}}mR_{ratio}^{\frac{5}{2}}}{\varepsilon_{en}})\end{aligned}$$

$$
\begin{aligned}
T_h &= 2n\Delta R_{ratio} \ln \frac{2n\Delta^2 R_{ratio} h_{max}^{(\infty)} \sqrt{n\Delta R_{ratio}}}{\delta_h} \\
&= 2n\Delta R_{ratio} \ln \left(2n\Delta^2 R_{ratio} h_{max}^{(\infty)} \sqrt{n\Delta R_{ratio}} \cdot \frac{16mR_{ratio}}{\varepsilon_{en} h_{max}^{(\infty)}}\right) \\
&= 2n\Delta R_{ratio} \ln \left(\frac{32\Delta^{\frac{5}{2}} n^{\frac{3}{2}} m R_{ratio}^{\frac{5}{2}}}{\varepsilon_{en}}\right)
\end{aligned}
$$

For $T_z = (2m \cdot \frac{\ln \delta_w^{-1}}{\delta_w} + 1) \cdot t_{\mathrm{mix}(B)}$

$$
T_z = (2m \cdot \frac{\ln \delta_w^{-1}}{\delta_w} + 1) \cdot t_{\mathrm{mix}(B)}
$$

Remember $t_{\mathrm{mix}(B)} = \lceil \log_2 (\frac{2m}{\delta_\xi}) \rceil \cdot \mathrm{mix}(B)$, $T_z = (2m \cdot \frac{-\ln \delta_\xi}{\delta_\xi} + 1) \cdot t_{\mathrm{mix}(B)}$, $\forall T_r = \frac{\ln\left(\frac{1}{12m}\delta_w\right)}{\ln \delta_\xi} T_z$.

For assumptions of Lemma 4.4.3 to Lemma 4.4.12 to be hold, we know there need to be $T_{conv} \geq t_{\mathrm{mix}(B)} + T_z + \max(T_h, T_y)$ and $T_{intl} \geq T_r + T_{conv}$.

Because $T_z \geq t_{\mathrm{mix}(B)}$ and $2T_y \geq T_h$, let $T_{conv} = 2T_z + 2T_y$, then we need

$$
\delta_P \leq \frac{\varepsilon_y}{2(3T_r + 2T_y)^2} \leq \frac{\varepsilon_y}{2(T_{conv})^2} \leq \frac{\varepsilon_{en}}{64\Delta n m R_{ratio}^2 (T_{conv})^2}
$$

which means

$$
\alpha \leq \frac{\varepsilon_{en}}{510 \cdot 32\Delta n m R_{ratio}^2 T_{conv}^3 \cdot \varepsilon}
$$

When we want to have an ε-energy approximation, we set $\varepsilon_{en} = \varepsilon$

$$
\begin{aligned}
\alpha &\leq \frac{\varepsilon}{510 \cdot 32\Delta n m R_{ratio}^2 T_{conv}^3 \cdot \varepsilon} \\
&= \frac{1}{510 \cdot 32\Delta n m R_{ratio}^2 T_{conv}^3}
\end{aligned}
$$

Then we fix T_{conv}. For $\delta_\xi = \frac{1}{30}$, $t_{\mathrm{mix}(B)} = O(\ln m)\mathrm{mix}(B)$, $T_z = O(m) \cdot t_{\mathrm{mix}(B)} = O(m \ln m)\mathrm{mix}(B)$,

$$
\begin{aligned}
T_{conv} &= 2T_z + 2T_y = O((m \ln m)\mathrm{mix}(B) + n\Delta R_{ratio} \ln(\frac{\Delta^{\frac{3}{2}} n^{\frac{3}{2}} m R_{ratio}^{\frac{5}{2}}}{\varepsilon_{en}})) \\
&= O(m \cdot \mathrm{mix}(B) + n\Delta R_{ratio})\mathrm{polylog}(nm/\varepsilon)
\end{aligned}
$$

$$
\begin{aligned}
T_r &= O(\ln\frac{1}{\delta_w})T_z = O(m\ln^2 m)\ln\frac{1}{\delta_w}\mathrm{mix}(B) \\
&= \frac{\Delta n m^2 R_{ratio}^2 (T_{conv})^2 \mathrm{mix}(B)\mathrm{polylog}(nm/\varepsilon)}{\varepsilon} \\
&= \frac{\Delta n^3 m^4 R_{ratio}^4 \mathrm{mix}(B)^3 \mathrm{polylog}(nm)}{\varepsilon}
\end{aligned}
$$

$T_{intl} = T_r + T_{conv} = O(T_r) = O(\frac{\Delta n^3 m^4 R_{ratio}^4 \mathrm{mix}(B)^3 \mathrm{polylog}(nm)}{\varepsilon})$.

Because the re-weighting algorithm runs in $T_{intl} + \frac{2\rho\ln m}{\alpha\varepsilon^2}$ time, the whole running time is $O(\frac{2\rho\ln m}{\varepsilon^2}\Delta n m R_{ratio}^2 T_{conv}^3 + \frac{\Delta n^3 m^4 R_{ratio}^4 \mathrm{mix}(B)^3 \mathrm{polylog}(nm)}{\varepsilon})$ time.

As $R_{ratio} = \frac{r_{max}^{(t)}}{r_{min}^{(t)}} \leq 6U_{max}^2 \frac{m}{\varepsilon}$ and we assume that the edge capacity U_{max} is $poly(n)$ and $\mathrm{mix}(B)$ is polynomial, then the running time is polynomial.

Let $\varepsilon_{en} = \varepsilon$, $\delta_\tau = \frac{1}{5}$, combine Lemma 4.4.2, Corollary 4.4.1 and Lemma 4.4.12, and we have the following theorem:

Theorem 4.4.1. *Our algorithm solves the following problem in Pure-LOCAL model in polynomial time:*

Weaker decision problem of Max-flow approximation. *For a graph $G = (V, E)$ with edge capacity U_e, $e \in E$ and a value F, if F does not exceed the $(1+O(\varepsilon))$ of the maximum flow, we will return "YES" . And if we return "NO" then F must exceed the maximum flow.*

More precisely, if $F \leq F^*$ then $cong_e(\bar{\boldsymbol{f}}) \leq 1$, and if $cong_e(\bar{\boldsymbol{f}}) > \frac{(1+\varepsilon)^2}{(1-\varepsilon)^2}$ then $F > F^*$.

The gap between 1 and $\frac{(1+\varepsilon)^2}{(1-\varepsilon)^2}$ is the case non-distinguishable by our Pure-LOCAL algorithm compared to the centralized algorithm.

4.5 Conclusions of Chapter

We studied the implementation of an multiplicative weights update algorithm approximating Max-flow problem in Pure-LOCAL model.

1. We design the way in Pure-LOCAL model to approximate the global quantity of average weight in a weight-varying iterative algorithm.

2. We implement in Pure-LOCAL model the computation to solving an electrical network with resistances changing in iterations. As we know, this is the first algorithm approximating the electrical flow in a weight-varying environment.

3. We show that our algorithm solves a weaker version of Max-flow approximation decision problem in polynomial time.

Chapter 5

Conclusion

We study three problems of different aspects of efficiency of algorithm design for large scale graph computations.

Firstly, we consider the question of generalizing modular decompositions and designing time-efficient algorithm for this problem. We present here positive results obtained for three definitions of modular decomposition in hypergraphs from the literature: the standard modules [73], the k-subset modules [13] and the Courcelle's modules [26]. We developed a general algorithmic scheme to compute their modular decomposition following the idea in [52] for modules in graphs, by implementing the appropriately functions in this scheme, we got a $O(n^3 \cdot l)$ algorithm for modular decomposition of standard module and k-subset module, improving the previous result based on a $O(n^4m^3)$ algorithm [74] for standard module and $O(n^{3k-5})$ algorithm [13] for k-subset module respectively, and conclude the decomposition of k-subset module in hypargraphs is in P. We also present negative results for two definitions of modular decomposition, allowing errors in graph modules that does not lead to a valid decomposition.

Another scenario we consider is the large scale graph data query. We study the generalized binary search problem [37] for which we compute an efficient query strategy to find a hidden target in graphs. We proposed a quasi-polynomial time approximation scheme for computing the optimal strategy of generalized binary search on weighted trees, which implies that the problem is not APX-hard, unless $NP \subseteq DTIME(n^{O(\log n)})$. By applying a generic reduction, we obtain as a corollary that the studied problem admits a polynomial-time $O(\sqrt{\log n})$-approximation. This improves previous $\hat{O}(\log n)$-approximation approaches, where the $\hat{O}$-notation disregards $O(\text{poly} \log \log n)$-factors.

Then we study the algorithm design on a distributed computing model with memory efficiency constraint, i.e. the Pure-LOCAL model. We studied the implementation of an multiplicative weights update algorithm [20] approximating Max-flow problem in Pure-LOCAL model. We design the way in Pure-LOCAL model to approximate the global quantity of average weight and approximately solving an electrical network in weight-varying iterations. To our knowledge, this is the first implementation of a weight update algorithm in Pure-LOCAL model. We show that our algorithm solves a weaker version of Max-flow $(1 + \varepsilon)$-approximation decision problem in polynomial time.

From a methodological point of view, the three types of efficiency concerns correspond to the following types of scenarios: the first one is the most classical one – given the problem, we try to design by hand the more efficient algorithm; the second one, the efficiency is regarded as an objective function – where we model query costs as an objective function, and using approximation algorithm techniques to get a good design of efficient strategy; the third one, the efficiency is in fact posed as a constraint of memory and we design algorithm under this constraint.

In the future, we'd like to explore more computation models that impose time or memory efficiency, and algorithm design on them. An interested question is: what cannot be computed if there are time or memory limitations? Recently there has been many studies on classical problems based on SETH conjecture, can we obtain hardness of polynomial time for hypergraph modules detection of decomposition? Can we build hardness results for memory limited computations in distributed model like Pure-LOCAL? This thesis presents our works in algorithm design purpose, but the complexity and hardness studies are also important and interesting for future works.

Bibliography

[1] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. Network flows. 1988.

[2] Ravindra K Ahuja, Thomas L Magnanti, James B Orlin, and MR Reddy. Applications of network optimization. *Handbooks in Operations Research and Management Science*, 7:1–83, 1995.

[3] Esther M. Arkin, Henk Meijer, Joseph S. B. Mitchell, David Rappaport, and Steven Skiena. Decision trees for geometric models. *Int. J. Comput. Geometry Appl.*, 8(3):343–364, 1998.

[4] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[5] Luca Becchetti, Vincenzo Bonifaci, Michael Dirnberger, Andreas Karrenbauer, and Kurt Mehlhorn. Physarum can compute shortest paths: Convergence proofs and complexity bounds. In *International Colloquium on Automata, Languages, and Programming*, pages 472–483. Springer, 2013.

[6] Luca Becchetti, Vincenzo Bonifaci, and Emanuele Natale. Pooling or sampling: Collective dynamics for electrical flow estimation. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1576–1584. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[7] Yosi Ben-Asher and Eitan Farchi. The cost of searching in general trees versus complete binary trees. Technical report, Technical report, 1997.

[8] Michael Ben-Or and Avinatan Hassidim. The bayesian learner is optimal for noisy binary search (and pretty good for quantum as well). In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 221–230, 2008.

[9] Claude Berge. *Graphes et hypergraphes*. Dunod, Paris, 1970.

[10] Louis J. Billera. Clutter decomposition and monotonic boolean functions. *Annals of the New-York Academy of Sciences*, 1970.

[11] Louis J. Billera. On the composition and decomposition of clutters. *Journal of Combinatorial Theory, Series B*, 11(3):234–245, 1971.

[12] Jan C. Bioch. The complexity of modular decomposition of boolean functions. *Discrete Applied Mathematics*, 149(1-3):1–13, 2005.

[13] Paola Bonizzoni and Gianluca Della Vedova. An algorithm for the modular decomposition of hypergraphs. *J. Algorithms*, 32(2):65–86, 1999.

[14] Paul S. Bonsma. The complexity of the matching-cut problem for planar graphs and other graph classes. *Journal of Graph Theory*, 62(2):109–126, 2009.

[15] Michele Borassi, Pierluigi Crescenzi, and Michel Habib. Into the square: On the complexity of some quadratic-time solvable problems. *Electr. Notes Th. Comp. Sci.*, 322:51–67, 2016.

[16] Yuri Y Boykov and M-P Jolly. Interactive graph cuts for optimal boundary & region segmentation of objects in nd images. In *Proceedings eighth IEEE international conference on computer vision. ICCV 2001*, volume 1, pages 105–112. IEEE, 2001.

[17] Renato Carmo, Jair Donadelli, Yoshiharu Kohayakawa, and Eduardo Sany Laber. Searching in random partially ordered sets. *Theor. Comput. Sci.*, 321(1):41–57, 2004.

[18] Pierre Charbit, Michel Habib, Vincent Limouzy, Fabien de Montgolfier, Mathieu Raffinot, and Michaël Rao. A note on computing set overlap classes. *Inf. Process. Lett.*, 108(4):186–191, 2008.

[19] M. Chein, Michel Habib, and M. C. Maurer. Partitive hypergraphs. *Discrete Mathematics*, 37(1):35–50, 1981.

[20] Paul Christiano, Jonathan A Kelner, Aleksander Madry, Daniel A Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 273–282. ACM, 2011.

[21] Vasek Chvátal. Recognizing decomposable graphs. *Journal of Graph Theory*, 8(1):51–53, 1984.

[22] Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Marco Molinaro. On the complexity of searching in trees and partially ordered structures. *Theor. Comput. Sci.*, 412(50):6879–6896, 2011.

[23] Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Caio Dias Valentim. The binary identification problem for weighted trees. *Theor. Comput. Sci.*, 459:100–112, 2012.

[24] Ferdinando Cicalese, Balázs Keszegh, Bernard Lidický, Dömötör Pálvölgyi, and Tomás Valla. On the tree search problem with non-uniform costs. *CoRR*, abs/1404.4504, 2014.

[25] Derek G. Corneil, H. Lerchs, and L. Stewart Burlingham. Complement reducible graphs. *Discrete Applied Mathematics*, 3(3):163–174, 1981.

[26] Bruno Courcelle. A monadic second-order definition of the structure of convex hypergraphs. *Inf. Comput.*, 178(2):391–411, 2002.

[27] Elias Dahlhaus. Parallel algorithms for hierarchical clustering and applications to split decomposition and parity graph recognition. *J. Algorithms*, 36(2):205–240, 2000.

[28] Dariusz Dereniowski. Edge ranking of weighted trees. *Discrete Applied Mathematics*, 154(8):1198–1209, 2006.

[29] Dariusz Dereniowski. Edge ranking and searching in partial orders. *Discrete Applied Mathematics*, 156(13):2493–2500, 2008.

[30] Dariusz Dereniowski and Marek Kubale. Efficient parallel query processing by graph ranking. *Fundam. Inform.*, 69(3):273–285, 2006.

[31] Dariusz Dereniowski and Adam Nadolski. Vertex rankings of chordal graphs and weighted trees. *Inf. Process. Lett.*, 98(3):96–100, 2006.

[32] Dariusz Dereniowski, Stefan Tiegel, Przemysław Uznański, and Daniel Wolleb-Graf. A framework for searching in graphs in the presence of errors. *arXiv preprint arXiv:1804.02075*, 2018.

[33] Peter G Doyle and J Laurie Snell. *Random walks and electric networks*, volume 22. American Mathematical Soc., 1984.

[34] Corel E., Lopez P., Meheust R., and Bapteste E. Network-thinking: Graphs to analyze microbial complexity and evolution. *Trends Microbiol.*, 24(3):224–237, 2016.

[35] Corel E, Meheust R, Watson AK, McInerney JO, Lopez P, and Bapteste E. Bipartite network analysis of gene sharings in the microbial world. *Mol. Biol Evol.*, 2018.

[36] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[37] Ehsan Emamjomeh-Zadeh, David Kempe, and Vikrant Singhal. Deterministic and probabilistic binary search in graphs. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 519–532, 2016.

[38] Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, 1994.

[39] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399?404, 1956.

[40] Jakub Gajarskỳ, Michael Lampis, and Sebastian Ordyniak. Parameterized algorithms for modular-width. In *International Symposium on Parameterized and Exact Computation*, pages 163–176. Springer, 2013.

[41] Tibor Gallai. Transitiv orientierbare Graphen. *Acta Mathematica Academiae Scientiarum Hungaricae*, 18:25–66, 1967.

[42] Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-optimal distributed maximum flow. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 81–90. ACM, 2015.

[43] Archontia C. Giannopoulou, Paul Hunter, and Dimitrios M. Thilikos. Lifo-search: A min-max theorem and a searching game for cycle-rank and tree-depth. *Discrete Applied Mathematics*, 160(15):2089–2097, 2012.

[44] Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[45] Michel Habib, Fabien de Montgolfier, and Christophe Paul. A simple linear-time modular decomposition algorithm for graphs, using order extension. In *SWAT, 9th Scandinavian Workshop on Algorithm Theory*, pages 187–198, 2004.

[46] Michel Habib, Ross M. McConnell, Christophe Paul, and Laurent Viennot. Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theor. Comput. Sci.*, 234(1-2):59–84, 2000.

[47] Michel Habib and Christophe Paul. A survey of the algorithmic aspects of modular decomposition. *Computer Science Review*, 4(1):41–59, 2010.

[48] Brent Heeringa, Marius Catalin Iordan, and Louis Theran. Searching in dynamic tree-like partial orders. In *Algorithms and Data Structures - 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings*, pages 512–523, 2011.

[49] Wen-Lian Hsu. Decomposition of perfect graphs. *Journal of Combinatorial Theory, Series B*, 43(1):70–94, 1987.

[50] Ananth V. Iyer, H. Donald Ratliff, and Gopalakrishnan Vijayan. Optimal node ranking of trees. *Inf. Process. Lett.*, 28(5):225–229, 1988.

[51] Ananth V. Iyer, H. Donald Ratliff, and Gopalakrishnan Vijayan. Parallel assembly of modular products – an analysis. Technical report, Technical Report 88-86, Georgia Institute of Technology, 1988.

[52] Lee O. James, Ralph G. Stanton, and Donald D. Cowan. Graph decomposition for undirected graphs. In *Proc. 3rd Southeastern International Conference on Combinatorics, Graph Theory, and Computing*, pages 281–290, 1972.

[53] Richard M. Karp and Robert Kleinberg. Noisy binary search and its applications. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 881–890, 2007.

[54] Meir Katchalski, William McCuaig, and Suzanne M. Seager. Ordered colourings. *Discrete Mathematics*, 142(1-3):141–154, 1995.

[55] Ioannis Koutis, Gary L Miller, and Richard Peng. A nearly-m log n time solver for sdd linear systems. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 590–598. IEEE, 2011.

[56] Eduardo Sany Laber, Ruy Luiz Milidiú, and Artur Alves Pessoa. On binary searching with nonuniform costs. *SIAM J. Comput.*, 31(4):1022–1047, 2002.

[57] Eduardo Sany Laber and Marco Molinaro. An approximation algorithm for binary searching in trees. *Algorithmica*, 59(4):601–620, 2011.

[58] Eduardo Sany Laber and Loana Tito Nogueira. Fast searching in trees. *Electronic Notes in Discrete Mathematics*, 7:90–93, 2001.

[59] Eduardo Sany Laber and Loana Tito Nogueira. On the hardness of the minimum height decision tree problem. *Discrete Applied Mathematics*, 144(1-2):209–212, 2004.

[60] Tak Wah Lam and Fung Ling Yue. Optimal edge ranking of trees in linear time. *Algorithmica*, 30(1):12–33, 2001.

[61] Yin Tat Lee, Satish Rao, and Nikhil Srivastava. A new approach to computing maximum flows using electrical flows. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 755–764. ACM, 2013.

[62] David A Levin and Yuval Peres. *Markov chains and mixing times*, volume 107. American Mathematical Soc., 2017.

[63] Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

[64] Nathan Linial and Michael E. Saks. Searching ordered structures. *J. Algorithms*, 6(1):86–103, 1985.

[65] Joseph W. H. Liu. Computational models and task scheduling for parallel sparse cholesky factorization. *Parallel Computing*, 3(4):327–342, 1986.

[66] Joseph W. H. Liu. The role of elimination trees in sparse factorization. *SIAM. J. Matrix Anal. & Appl.*, 11(1):134–172, 1990.

[67] László Lovász et al. Random walks on graphs: A survey. *Combinatorics, Paul erdos is eighty*, 2(1):1–46, 1993.

[68] Aleksander Madry. Computing maximum flow with augmenting electrical flows. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 593–602. IEEE, 2016.

[69] Ross M. McConnell. A certifying algorithm for the consecutive-ones property. In *SODA, Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 768–777, 2004.

[70] Ross M. McConnell and Fabien de Montgolfier. Linear-time modular decomposition of directed graphs. *Discrete Applied Mathematics*, 145(2):198–209, 2005.

[71] Augustine M. Moshi. Matching cutsets in graphs. *Journal of Graph Theory*, 13(5):527–536, 1989.

[72] Shay Mozes, Krzysztof Onak, and Oren Weimann. Finding an optimal tree searching strategy in linear time. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 1096–1105, 2008.

[73] R.H. Möèring and F.J. Radermacher. Substitution decomposition for discrete structures and connections with combinatorial optimization. *Proceedings of the Workshop on Algebraic Structures in Operations Research*, pages 257–355, 1984.

[74] Rolf H. Möèring. Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and boolean functions. *Annals of Operation Research*, 4:195–225, 1985.

[75] Jaroslav Nesetril and Patrice Ossona de Mendez. Tree-depth, subgraph coloring and homomorphism bounds. *Eur. J. Comb.*, 27(6):1022–1041, 2006.

[76] Krzysztof Onak and Pawel Parys. Generalization of binary search: Searching in trees and forest-like partial orders. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 379–388, 2006.

[77] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[78] Alex Pothen. The complexity of optimal elimination trees. Technical report, Technical Report CS-88-13, Pennsylvannia State University, 1988.

[79] Ronald L. Rivest, Albert R. Meyer, Daniel J. Kleitman, Karl Winklmann, and Joel Spencer. Coping with errors in binary search procedures. *J. Comput. Syst. Sci.*, 20(3):396–404, 1980.

[80] Irena Rusu and Jeremy Spinrad. Forbidden subgraph decomposition. *Discrete mathematics*, 247(1-3):159–168, 2002.

[81] Alejandro A. Schäffer. Optimal node ranking of trees in linear time. *Inf. Process. Lett.*, 33(2):91–96, 1989.

[82] Alexander Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer Science & Business Media, 2003.

[83] Paolo Serafino. Speeding up graph clustering via modular decomposition based compression. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 156–163, 2013.

[84] Daniel A Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the STOC*, volume 4, 2004.

[85] George Steiner. Searching in 2-dimensional partial orders. *J. Algorithms*, 8(1):95–105, 1987.

[86] Jayme Luiz Szwarcfiter, Gonzalo Navarro, Ricardo A. Baeza-Yates, Joísa de S. Oliveira, Walter Cunto, and Nivio Ziviani. Optimal binary search trees with costs depending on the access paths. *Theor. Comput. Sci.*, 290(3):1799–1814, 2003.

[87] Marc Tedder, Derek G. Corneil, Michel Habib, and Christophe Paul. Simpler linear-time modular decomposition via recursive factorizing permutations. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, pages 634–645, 2008.

[88] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.