Quasi-Isometric Graph Simplification and Graph Centrality

Roger Chi-Wei Su

(Khí-Uí Soo)

A thesis submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy (PhD)
in Computer Science
at the University of Auckland, 2022.

Abstract

This thesis studies some theoretical aspects of graph simplifications. Graph simplifications are methods to transform a graph into a smaller representation such that computation on the simplified representation becomes more efficient while remaining reasonably accurate. The distance query is a common computation task on graphs, and we introduce the concept of quasi-isometries as a measure of the distance distortions of graph simplifications.

We also introduce a generic graph simplification method called partition-graphs. This method constructs a new graph based on a partition of the original vertices into subsets that induce connected subgraphs. We establish that a partition-graph is quasi-isometric to the original graph, with the quasi-isometry's parameters depending on the diameters of the subsets.

In addition to studying quasi-isometries and distance distortions, we investigate the preservation of two centrality concepts: the centre and the median. Although we have found partition-graphs of trees that do not preserve the centres and medians, we show specific method of constructing partition-graphs of a tree that preserves the centre or the median.

Finally, we focus on the set of all possible partition-graphs of a path, which are called partition-paths and can be viewed as integer compositions with restricted part-sizes. Under three specific settings, we use combinatorial techniques to derive a recurrence to count the total number of balanced partition-paths, and establish regular-language characterisations of the possible outcomes of two probabilistic path-partitioning algorithms.

Acknowledgment

The author would like to thank:

- Prof. Bakhadyr Khoussainov the main supervisor, Dr. Simone Linz the co-supervisor, and AProf. Jing Sun the doctoral co-ordinator in the School of Computer Science.
- Ms. Robyn Young the secretary in the School of Computer Science, Mr. Bothwell Wong the digital technician, and the various administration and support teams across the university.
- Many staff across the Faculty of Science for other academic discussions: Dr. Michael Dinneen, AProf. Simon Harris, Prof. Vivien Kirk, Dr. Jiamou Liu, Prof. André Nies, Dr. Miao Qiao, Dr. Jeroen Schillewaert, and Dr. Mark Wilson.
- The University of Auckland Doctoral Scholarship.
- His fellow doctoral students, friends outside university, and Dr. Lucy Han the dentist.
- The eateries and cafés around central Auckland—special thanks to Megumi at AUT's friendly Refuel canteen, Kyung-Min and his cozy café, and the two *rāmen* houses for the occasional extra slices of *chāshū* pork.
- All his teachers at the University of Waikato who first opened the author's eyes to the boundless world of knowledge and inspired him to embark on the academic path.
- His wonderful family thousands of kilometres away, for supporting the author through the periods of doubt, the sleepless nights, and the ruthless lockdowns.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

O muse, o alto ingegno, or m'aiutate;
o mente che scrivesti ciò ch'io vidi,
qui si parrà la tua nobilitate.

—Dante (Inferno II.7–9) [1]

Simplification of data is an important task in many aspects of life, particularly in the field of computing. Simplification is the removal of redundant information, so that by focusing on the remaining part of the data, computations can be made feasible or more efficient. Developing good simplification methods is a challenging task—a good simplification method not only should efficiently remove as much redundancy as possible, but should also keep the parts of the data that are crucial to the specific scenario.

The motivation behind this thesis is simplification of the prevalent data-type of *graphs*. A graph is a mathematical structure consisting of *vertices* that are interconnected by *edges*, and is useful for modelling abstract relationships between objects.

An example of a graph is academic citations. Here the objects of interest are a set of academic publications, with citations being the relationships of interest. When article A cites

[1] Oh Muse, oh high ingenuity, now help me; / oh memory, who wrote down what I saw, / here is where your nobility will manifest.

book B, a directed edge is placed from vertex A to vertex B in the graph. (See Figure 1.1a.) Then one may want to compute queries such as: which publications are cited the most, or whether there is a pair of publications that cite each other. Questions like these can be nicely handled once the problem is modelled as a graph. In graph-theoretic terms, the most-cited publications are the vertices with the maximum in-degree (the number of incoming edges), while a pair of publications citing each other is called a 2-cycle.

Webpages and hyperlinks can also be modelled in a similar way. An edge from vertex x to vertex y can mean that webpage x contains a hyperlink to webpage y. Now a possible question is: whether there is a collection of webpages such that one can move freely between them by clicking on hyperlinks only. Such a set of webpages is in fact called a strongly connected component in graph-theoretic terms.

The two examples above can be considered real-life scenarios, but graphs can model more abstract scenarios too. Consider discrete-event systems used to control factories or traffic lights, for instance. The set of all configurations of a discrete-event system can be viewed as the vertices, with an edge from vertex q_1 to vertex q_2 meaning that it is possible for the system to change from configuration q_1 to configuration q_2.

The examples so far demonstrate asymmetric relationships, modelled by *directed graphs*, in which each edge has a direction. However, symmetric relationships are abundant too. A physical example is a computer network, where the computers in a building may be connected by cables, so here the vertices are the computers, and two directly connected computers are

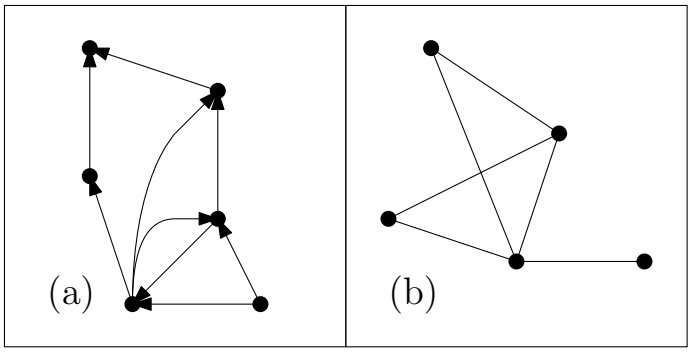


Figure 1.1: Examples of directed and undirected graphs.

joined by an undirected edge. (See Figure 1.1b.) Meanwhile, an abstract example is event scheduling, where a set of events can be viewed as vertices, and two vertices are joined by an undirected edge when the corresponding events overlap in time.

These graphs with undirected edges are called *undirected graph*, or simply a *graph*. Note that every undirected graph is also a directed graph, with each undirected edge representing two directed edges going in both directions. A typical problem in the networking application is: how to route a message from one computer to another as quickly as possible. In a more detailed model, each edge can have an associate cost. Then one can ask: how to route a message from computer C_1 to computer C_2 with the lowest cost.

Once the situation of interest is modelled as a graph, one can focus on the abstract structure and seek algorithms to answer many questions about the graph. For example: whether the graph is connected; what is the number of connected components; how far apart are two given vertices; from which vertices can one move to any other vertex as quickly as possible.

Relating back to the first example of academic citations, the most-cited publications are the vertices with the largest in-degree, and a pair of publications citing each other is a directed cycle of length two. Both of these questions can be computed efficiently on a directed graph by traversing all the vertices only once. In the example of webpages, a freely navigable set of webpages corresponds to the concept of a *strongly connected component* in a directed graph, which has a well-established *depth-first search* algorithm [31, Section 22.5]. As for the example of computer networks, the lowest-cost route from computer C_1 to computer C_2 is exactly the shortest-path problem on an undirected graph with positive edge-weights; distance queries like these are very important, and are discussed in more detail below.

The distance query. The distance query is among the most common types of computation on graphs. Internet routing is the most prevalent example; to transmit information, a

device needs to know where is the best next step to send the packets, so that the latency is minimised. In physical route-planning, we often want to know how to get from one location to another with the smallest amount of effort (combining a range of different metrics, including the physical distance as well as the gradient[2]).

In a directed graph, the distance from one vertex v_1 to another vertex v_2 is defined to be the length of a shortest path from v_1 to v_2. Hence, behind the distance query is the shortest path problem, which has several variants:

(a) compute a shortest path from v_1 to v_2,

(b) compute the distance from v_1 to v_2 (a numeric answer only),

(c) compute the distances from v_1 to all other vertices,

(d) compute the distances between all the possible pairs v_1 to v_2.

Apart from these direct queries, there are many other graph parameters based on the distance. For example, the diameter of a graph is the maximal value among all the possible distances, and it plays a central role in the field of Network Design [85]. Another example is the various concepts of the central locations in a graph; many of these centrality concepts are defined in terms of the distance, and are relevant to Operational Research[3] [49, 107]. Computing these parameters requires computing distances first, which demonstrates the importance of efficient algorithms for the distance query.

One of the earliest shortest path algorithms is Dijkstra's algorithm [31, Section 24.3], which computes the distances from a single source-vertex on a directed graph with positive edge-weights. Dijkstra's algorithm can compute a shortest path from a source-vertex to a target-vertex, solving variants (a) and (b) above. It can also compute the distances from a source-vertex to every other vertex, which is variant (c). These variants deal with only one source-vertex, and hence are known as the *single-source shortest path* problems.

[2] In the hilly city of Auckland where this thesis was written, the gradient is an important factor indeed.
[3] Also called *Operations Research*.

The Bellman-Ford algorithm [31, Section 24.1] is another algorithm for the single-source shortest path problems. Given a source-vertex in a directed graph with possibly negative edge-weights (without negative-weight cycle), the Bellman-Ford algorithm solves variant (c).

There are situations where we want to query the distances so often that it is inefficient to perform a computation every time a query is received. Under these situations, it is better to compute the table of all the distances at the start. This is known as the *all-pair shortest path* problem, which is variant (d), and a well-known solution is the Floyd-Warshall algorithm [31, Section 25.2].

The classical algorithms above were the first solutions to their respective problems, and their solutions are exact. However, despite their running times are considered theoretically efficient, they often become inadequate for applications. For instance, Dijkstra's algorithm's worst-case running time is asymptotically $\Theta(m + n \log n)$. Theoretical Computer Science considers this a good bound, since it is not exponential. Yet, faced with a graph on millions of vertices, classical algorithms like Dijkstra's do not have a good practical running time. In the Koblenz Network Collection [70], the graph of webpages and hyperlinks in the `.uk` domain in the year 2002 already has over 18 million vertices, and it is reasonable to expect real-life graphs nowadays to be significantly more enormous as well as more numerous.

Techniques for large-scale graphs. There are two possible remedies to this issue with scalability. One is to optimise the classical algorithms or to design new ones, devising more and more intricate procedures or data structures. An example of this line of research on the distance is the work on 2-hop labels [30].

Another remedy meanwhile trades some accuracy for more efficiency. This is a useful approach for many computational problems, including those studied by the substantial field of Approximation Algorithms [31, Chapter 35]. Even though some accuracy is sacrificed, it is important to derive theoretical bounds that guarantee that the results is still reasonably accurate.

Large-scale graphs naturally contain much information that is redundant for the specific computations. As an hypothetical example, when computing the number of connected components in a graph, the details of the individual edges are not that important. Once we know that four vertices are in the same connected component, we no longer need to concern how exactly they are linked. If the graph is preprocessed by removing some edges in a correct way, we can still correctly compute the number of connected components but much quicker.

This line of effort can be collectively termed Graph Simplification. For a specific computation problem, we summarise the useful information in the graph and discard the redundant parts. At the end we have a smaller graph that is easier for comprehension or computation.

One family of graph simplification methods involves edge-removals. Some edges do not contribute to the shortest paths, and some do not affect the connectivity much. Removing the first type of edges preserves the distance properties, while deleting the second type preserves the connectivity. These methods respectively correspond to *spanners* [94] and *sparsifiers* [112], which Section 3.1 will describe in more detail.

Vertex-grouping is another operation that serves as the basis of an important family of graph simplification methods. A major topic is Graph Clustering[4] where the objective is to find the close-knit clusters in a large graph. (See Figure 1.2.) Later we will propose a new simplification framework that uses vertex-grouping but is based on the distance instead of clusters.

Distance-approximation and quasi-isometries. Earlier we mentioned that the distance query is among the most important computational tasks on graphs. When faced with a large-scale graph, one can simplify or preprocess it, so that that distance can be computed more efficiently, while the answer is in an approximate range.

In established fields, the distance-approximation property comes in similar forms. For instance, the early works on spanners [94] and distance oracles [117] stipulate that for every pair of vertices, the quotient of the new distance over the original distance must be bounded

[4]Also known as *Graph Partitioning* and *Community Detection*.

by a constant. Formally, let G be the original graph and H be the simplified graph or the preprocessed structure. For every two vertices v_1 and v_2 in G, let $d_G(v_1, v_2)$ denote their distance in G, and we assume that their distance in H, denoted $d_H(v_1, v_2)$, is also well-defined. Then the distance-approximation property in the early works [94, 117] has the form

$$d_H(v_1, v_2) \leq \alpha \cdot d_G(v_1, v_2),$$

where α is a constant. Subsequent works then generalised this inequality by allowing an extra additive constant β, resulting in inequalities of the form

$$d_H(v_1, v_2) \leq \alpha \cdot d_G(v_1, v_2) + \beta.$$

These are for the simplification or preprocessing methods that increase the distances. Meanwhile there are also situations, such as [15], where the distances are decreased. Overall, these similar inequalities turn out to be captured by the general notion called *quasi-isometries*.

A quasi-isometry is a mapping φ from a graph G to another graph H, such that for all v_1 and v_2 in G, the distortion between $d_G(v_1, v_2)$ and $d_H(\varphi(v_1), \varphi(v_2))$ are bounded by an additive constant and a multiplicative constant. More precisely, a quasi-isometry has a positive-integer parameter A and a non-negative-integer parameter B such that the following

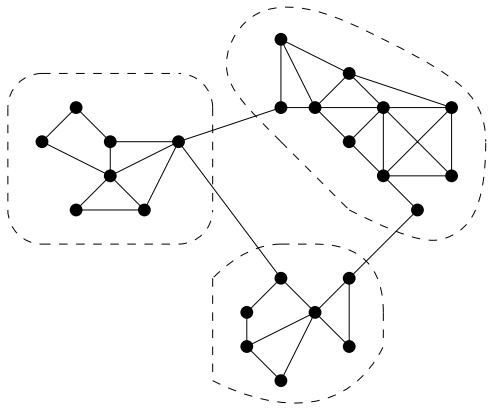


Figure 1.2: Example of graph clustering.

quasi-isometry inequality holds:

$$\frac{1}{\alpha} \cdot d_G(v_1, v_2) - \beta \leq d_H(\varphi(v_1), \varphi(v_2)) \leq \alpha \cdot d_G(v_1, v_2) + \beta.$$

The smaller the constants, the better the distance preservation. Therefore, we always strive for quasi-isometries with small constants.

Quasi-isometries were conceived in algebraic geometry, where they are used to classify infinite mathematical objects. Introducing them into the realm of finite graphs is the first novelty of this thesis.

Diameter-bounded vertex-grouping. Another motivating idea behind this thesis is vertex-grouping based on distance. Earlier we mentioned that vertex-grouping is mainly employed in Graph Clustering, where the main goal is to maximise some metric that measure how 'close-knit' the clusters are. One example of such metrics is the ratio between the number of edges within a vertex-group and the number of edges between different vertex-groups.

In Graph Clustering, the diameter of each cluster is not the main concern. However, there is another intuition, illustrated by Figure 1.3. Suppose we zoom out from a graph, then the individual vertices become blurred, and we end up seeing a new graph whose vertices are uniform-size blobs that contain original vertices. A key property in this intuition is that every two vertices in a vertex-group are not too far apart.

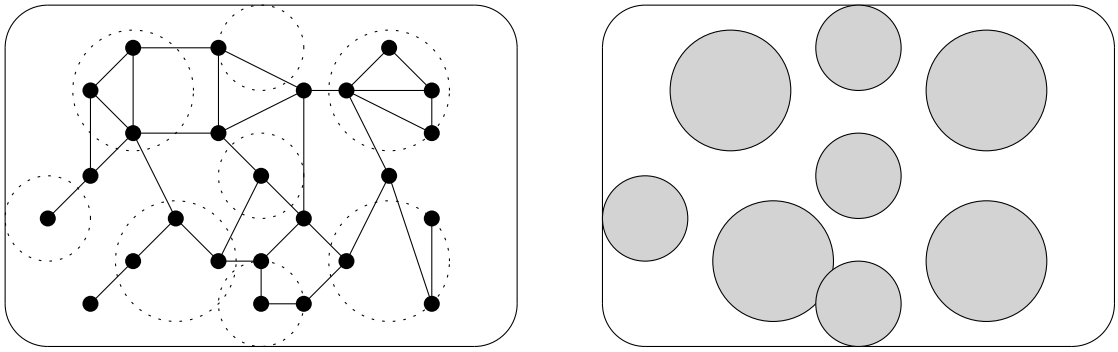


Figure 1.3: Intuition behind diameter-bounded vertex-grouping.

This intuition motivates the idea of *diameter-bounded vertex-grouping*, where we assign vertices into a group not by how strongly connected they are, but by how *close* they are. This is our second motivation, which happens to be related to the first idea of quasi-isometries in the sense that they are both distance-related.

Partition-graphs. As a start to our investigation of quasi-isometries and diameter-bounded vertex-grouping, we propose a general framework of building simplified graphs from a partition of the vertices.

In the clustering context, vertex-grouping outputs a set of connected vertex-subsets only. But in addition to just grouping vertices, we can construct a new graph called a *partition-graph*, whose vertices are these groups. To distinguish between the original graph and the partition-graph, the vertices and edges of the partition-graph are respectively called *super-vertices* and *super-edges*. Two super-vertices V_1 and V_2 in the partition-graph are joined by a super-edge when there is at least one edge linking a vertex in V_1 and a vertex in V_2. Partition-graphs, as well as quasi-isometries in the next paragraph, will be more formally defined later in Chapter 5.

To derive a quasi-isometry inequality for a partition-graph, we need to use an upper bound on the diameters of the super-vertices. Let H be a partition-graph of G. If every super-vertex in H has diameter no more than a natural number c, then the quasi-isometry from H to G has constants $A = c + 1$ and $B = 1$. This will be discussed in detail by Theorem 5.9.

Partition-graphs and centrality. Now that the distance function of a partition-graph can be approximately preserved in the form of a quasi-isometry inequality with small constants, we then move on to other distance-related properties that may be preserved by the construction of partition-graphs. In this thesis we chooses the two most basic centrality notions as the first step: the *centre* and the *median*, both of which are defined in terms of distances, and are surveyed by Chapter 4.

One of the purposes of graph simplification is to make computations feasible. Suppose G is a large-scale graph, whose centre C_G is impractical to locate. Then we seek a simplified graph H (with a small-constant surjective quasi-isometry $\varphi : G \to H$), such that the centre of H (denoted C_H) can be efficiently located. Next, we want to quantify the accuracy with which we can infer the centre of G using C_H and the mapping φ. The natural way is to find the vertices in G that is mapped to C_H by φ. This set is called the reverse image, and is denoted $\varphi^{-1}(C_H)$. Now, the accuracy of the approximation is $d_G(C_G, \varphi^{-1}(C_H))$, the distance between the actual centre of G and the inferred $\varphi^{-1}(C_H)$. This value is called the *centre-shift*, which is defined and explored in Section 5.3. When the centre-shift of a graph simplification is zero, we say that the simplification preserves the centre.

In Section 5.3, we also derive a theorem which shows that a quasi-isometric simplification alone does not effectively yield an upper bound on the centre-shift.

Although general quasi-isometric simplifications do not always preserve the centre by having a small centre-shift, there are still particular ways to construct partition-graphs on trees such that the centre and the median are exactly preserved. To develop these specific constructions, we require some basic facts on the centres and medians in trees, so we devote Chapter 4 to a survey of old and new results. With the utilities in Chapter 4, we then develop a method to construct centre-preserving partition-trees in Section 5.4, and another method to build partition-trees that preserve the median in Section 5.5.

All partitions of a graph. A graph can corresponds to multiple partition-graphs, even when we restrict the diameters of the super-vertices. Hence, given a graph and a diameter-bound, it is desirable to study the properties shared by all possible partition-graphs; example properties include the maximal, minimal or average number of super-vertices in the partition-graphs of a fixed diameter-bound.

Nevertheless, such goals are challenging on trees, let alone general graphs, so we begin with path-graphs. In a partition-graph of a path-graph, the diameter and the cardinality of

a super-vertex become equivalent notions. Furthermore, partition-graphs of P_n (the path-graph on n vertices) can be viewed as integer compositions of n with restricted part-sizes, and this combinatorial flavour makes the reasoning simpler.

In Chapter 6, we start by considering all the partition-graphs of P_n with every super-vertex having diameter either two or three. We derive recurrences that count these partition-graphs of P_n, and also study a balance property that is related to the centre-shift.

On a general graph, one way of constructing a partition-graph is to pick an unassigned vertex v at random, group v and its unassigned neighbours into a new super-vertex, and repeat until every vertex is assigned to a super-vertex. In a resultant partition-graph, every super-vertex has diameter one, two or three. The second part of Chapter 6 applies this algorithm to path-graphs, and characterises the set of all possible outcomes on P_n as the set of sequences over $\{1, 2, 3\}$ that sum to n and avoid the regular expression 12^*1.

A partition-graph is not an effective simplification when it contains size-one super-vertices. Hence, in the third part of Chapter 6 we modify the previous method into an algorithm that works specifically on P_n and creates a partition-graph with the super-vertices' diameters ranging from two to five. Similar to the previous method, we characterises the set of all possible outcomes on P_n as the set of sequences over $\{2, 3, 4, 5\}$ that sum to n and avoid certain patterns.

1.1 Summary and Thesis Organisation

To summarise, this thesis is motivated by two new ideas.

The first is on quasi-isometries, which were originally used on infinite mathematical objects only. In Chapter 5 we introduce quasi-isometries to the realm of finite graphs, and explore how they could be used as a general measure of distance-approximation.

The second motivating question behind this thesis is on vertex-grouping. In the context of Graph Clustering, the most notable field that employs vertex-grouping, the main objective is

the close-knit property of the vertex-groups. However, earlier we described another 'zooming out' intuition, where the vertex-groups should be constructed with the principle that every two vertices in a group should not be too far apart. This motivates the idea of diameter-based vertex-grouping, which will be explored in more detail in Chapter 5.

This thesis is structured as follows.

To start with, Chapter 2 lays out some preliminary concepts, which will become helpful when Chapter 3 surveys the background works.

The centre and the median are the main foci of our graph simplification methods, so Chapter 4 summarises results on the centres and medians in trees. These results will come to use in Sections 5.4 and 5.5. The final part of Chapter 4 also briefly discusses the structural properties of the centre of a k-tree.

Chapter 5 is the focal part of this thesis. Section 5.1 presents quasi-isometries, and Section 5.2 formally defines partition-graphs. Section 5.3 introduces the centre-shift, and establishes the weak upper bound on the centre-shift implied by a quasi-isometry alone. Next, Section 5.4 presents *outward-contraction*, the method to construct a partition-graph on a tree such that the centre is preserved, and Section 5.5 shows that storing the cardinality of every super-vertex allows any partition-graph of a tree to preserve the median.

Next, Chapter 6 devotes to partition-graphs on paths. Section 6.1 studies the recurrences related to the set of partition-graphs of P_n with every super-vertex having diameter two or three. Section 6.2 studies the first randomised partition-graph algorithm on P_n, and characterises the set of possible output in terms of a regular language. Section 6.3 studies an augmented randomised partition-graph algorithm on P_n, and characterises the set of possible output in terms of another regular language.

Finally this thesis closes with the epilogue Chapter 7.

Chapter 2

Preliminaries

This chapter outlines some basic definitions related to graphs, graph classes and integer compositions. But we cover some general terminology first. The set of natural numbers $\mathbb{N}$ is $\{1, 2, \ldots\}$, the set of non-negative integers $\mathbb{N}_0$ is $\{0, 1, 2, \ldots\}$, and $[n]$ denotes $\{1, 2, \ldots, n\}$ for some $n \in \mathbb{N}$.

Let φ be a function from a set A to a set B. For a subset $S \subseteq B$, the *reverse image* of S is denoted by $\varphi^{-1}(S)$, and consists of all the elements in A that are mapped to S. In other terms,

$$\varphi^{-1}(S) = \{a \in A \mid \varphi(a) \in S\}.$$

2.1 Basics of Graphs

A *graph* G is made up of a finite set of *vertices* $V(G)$ and a set of *edges* $E(G)$, where an edge is a 2-element subset of $V(G)$.

In applications, the vertices represent the objects of interest, and the edges represent the relationships between these objects. In this setting, these relationships are all symmetric. Non-symmetric relationships are modelled by directed graphs, and some applications use graphs with parallel edges, but these are not our main focus.

Two vertices v_1 and v_2 are said to be *adjacent* when $\{v_1, v_2\} \in E(G)$, and this is denoted

by $v_1 \sim v_2$. A vertex is *incident to* an edge when it is one of the two endpoints of this edge. On the other hand, an edge $\{v_1, v_2\}$ is *incident on* both vertices v_1 and v_2. The *degree* of a vertex v, denoted $\deg(v)$, is the number of edges incident on v.

Most of the time, we simply use G to mean $V(G)$ when there is no ambiguity. By convention, the number of vertices and the number of edges are denoted by n and m respectively.

The operation of *deleting* a vertex u from a graph G produces another graph G' with $V(G') = V(G) \setminus \{u\}$, and $\{v_1, v_2\} \in E(G')$ if and only if $v_1 \neq u$ and $v_2 \neq u$. In other words, u is deleted from the vertex-set of G, and all the edges incident on u are removed.

Subgraphs. When it comes to the sub-structure of a graph, there are three related concepts.

A *subgraph* H of a graph G is one that satisfies $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. In other words, we can obtain a subgraph by deleting some vertices (together with their incident edges) and then deleting some additional edges.

A *spanning subgraph* H of a graph G is one that satisfies $V(H) = V(G)$ and $E(H) \subseteq E(G)$. In other words, a spanning subgraph is obtained by deleting edges only.

An *induced subgraph* H of a graph G is one that satisfies $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$, with the extra criterion that for all $v_1, v_2 \in V(H)$, $\{v_1, v_2\} \in E(G)$ implies $\{v_1, v_2\} \in E(H)$. In other words, an induced subgraph is obtained by deleting vertices (and their incident edges) only, and no extra edge can be removed.

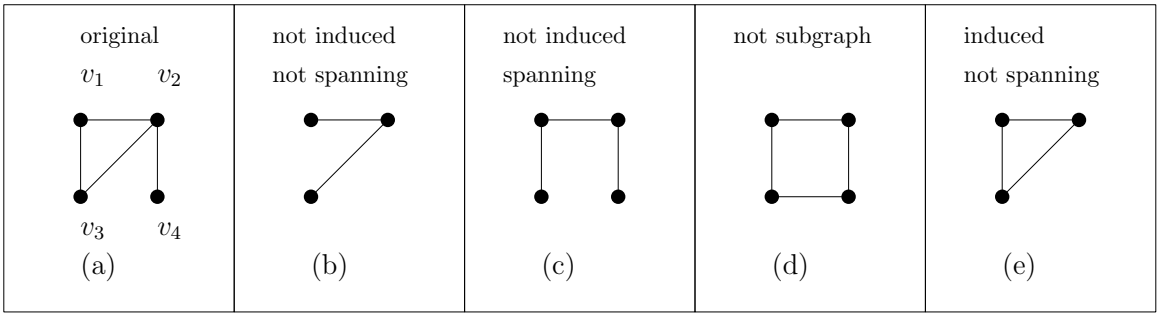


Figure 2.1: Examples of subgraphs, spanning subgraphs and induced subgraphs of (a).

In Figure 2.1, (a) is the original graph, while (b) to (e) are various related graphs.

(b) This is just a subgraph, and is neither spanning nor induced. It is not spanning because v_4 is not present, and not induced because the edge $\{v_1, v_3\}$ is absent.

(c) This is a spanning subgraph because every vertex in (a) is present, but is not an induced subgraph because the edge $\{v_2, v_3\}$ is absent.

(d) This is not a subgraph at all because the edge $\{v_3, v_4\}$ does not belong to the original graph in (a).

(e) This is an induced subgraph, but is not spanning because v_4 is not present.

Paths. A *path* is a sequence of vertices $[v_1, v_2, \ldots, v_l]$ such that $v_i \sim v_{i+1}$ or $v_i = v_{i+1}$ for all $i \in [l-1]$. A *simple path* is a path in which every vertex is distinct. A *cycle* is a path that starts and ends at the same vertex; that is, when $v_1 = v_l$. The length of a path is the number of edges in the path; when a path has l vertices, its length is $l - 1$.

Two vertices v_1 and v_2 are said to be *connected* when there is a path with v_1 and v_2 as the endpoints. A graph is said to be *connected* when every two vertices are connected. For $k \in \mathbb{N}$, a *k-connected* graph is one that has more than k vertices and remains connected when fewer than k vertices are removed.

2.2 Distance-Related Concepts

In a connected graph, the *distance* between two vertices v_1 and v_2, denoted by $d(v_1, v_2)$, is defined to be the length of the shortest path between them. In particular, the distance from a vertex to itself is always 0, i.e. $d(v, v) = 0$, since the 1-vertex path $[v]$ has length 0. The assumption of the graph's connectedness is crucial; when two vertices are not connected, their distance is not well-defined.

Now that we have defined the distance, we continue with some distance-related concepts.

The *eccentricity* of a vertex v is the maximum distance from v to any other vertex:

$$\mathrm{ecc}(v) = \max\{d(v, x) \mid x \in G\}.$$

The *eccentricity-witnesses* (*ecc-wits* for short) of a vertex v are the vertices x such that $\mathrm{ecc}(v) = d(v, x)$. Furthermore, the *diameter* of G is the maximum eccentricity:

$$\mathrm{diam}(G) = \max\{\mathrm{ecc}(x) \mid x \in G\},$$

and the *radius* of G is the minimum eccentricity:

$$\mathrm{rad}(G) = \min\{\mathrm{ecc}(x) \mid x \in G\}.$$

A *diameter-path* is a path whose length equals the diameter.

The distance between vertices can be extended to the distance between vertex-subsets. For two subsets $S_1, S_2 \subseteq G$, their distance $d(S_1, S_2)$ is defined to be

$$\min\{d(x_1, x_2) \mid x_1 \in S_1, x_2 \in S_2\}.$$

According to this definition, the distance between two overlapping vertex-subsets is always 0. In a similar fashion, for a vertex $v \in G$ and a vertex-subset $S \subseteq G$, we define $d(v, S)$ to be

$$\min\{d(v, x) \mid x \in S\}.$$

Weighted graphs. Often, a relationship between two objects has an associated numerical value. Such a value can represent the bandwidth of the connection between two computers, or the distance between two geographical locations. These situations are described by the notion of a *weighted graph*, which consists of a vertex-set $V(G)$, an edge-set $E(G)$ and a weight-function $w : E(G) \to \mathbb{R}^+$ that assigns a positive real number to each edge. For an

edge $\{v_1, v_2\}$, the value $w\big(\{v_1, v_2\}\big)$ is called the weight of this edge.

One could allow the weights to be non-negative or any real number, but some technical complications might arise when weights are zero or negative. Most applications do not use zero or negative weights anyway, so this thesis always assumes the weights to be positive real numbers.

In a weighted graph, the length of a path is the sum of the weights of the path's edges. In other terms, if the path is $[v_1, v_2, \ldots, v_l]$, then its length is

$$\sum_{i=1}^{l-1} w\big(\{v_i, v_{i+1}\}\big).$$

For any two vertices v_1 and v_2, their distance $d(v_1, v_2)$ is defined to be the length of the shortest path between them.

An unweighted graph can be viewed as a weighted graph with every edge having weight one. When a graph does not have the adjective 'weighted' in front, we consider it to be unweighted.

2.3 Metric Spaces

A *metric space* consists of a set M and a *distance* or *metric* function d that maps from $M \times M$ to the non-negative real numbers. The function d must satisfy the following three axioms for all $v_1, v_2, v_3 \in M$:

(M1) $d(v_1, v_2) = 0$ if and only if $v_1 = v_2$

(M2) $d(v_1, v_2) = d(v_2, v_1)$

(M3) $d(v_1, v_3) \leq d(v_1, v_2) + d(v_2, v_3)$

The third axiom (M3) is commonly known as the *triangle inequality*.

Every connected weighted graph can be viewed as a metric space, as the distance function defined in the previous paragraph satisfies these three metric-space axioms. The first two

axioms (M1) and (M2) trivially hold in every connected graph, while the third axiom (M3) is verified below.

Proof of (M3) in a graph. Suppose $d(v_1, v_2) + d(v_2, v_3) < d(v_1, v_3)$. By definition, $d(v_1, v_3)$ is the length of the shortest path from v_1 to v_3. However, the assumption implies that there is a shorter path from v_1 through v_2 to v_3, which means that $d(v_1, v_3)$ is not the length of the shortest path from v_1 to v_3. Hence, $d(v_1, v_2) + d(v_2, v_3)$ cannot be less than $d(v_1, v_3)$. □

It is then important to have a notion that expresses the situation when two metric spaces have the same distance function. Let M_1 and M_2 be two metric spaces, with respective distance functions d_1 and d_2. A mapping $f : M_1 \to M_2$ is called an *isometry* if for every $x, y \in M_1$:

$$d_1(x, y) = d_2(f(x), f(y)).$$

Note that an isometry is necessarily injective. If $x \neq y$, then (M1) implies $d_1(x, y) > 0$. Now $d_2(f(x), f(y)) > 0$ also holds because f is an isometry. This latter inequality means that $f(x) \neq f(y)$, which completes the reasoning that f is injective.

2.4 Basics of Centre and Median

This section lays out the basic definitions of the *centre* and the *median*, which are the two centrality notions studied later parts of this thesis.

Centre. Recall from Section 2.2 that the eccentricity of a vertex $v \in G$ is defined to be $\mathrm{ecc}(v) = \max\{d(v, x) \mid x \in G\}$. In addition, the radius is the smallest eccentricity value in a graph, and the diameter is the largest eccentricity value. Then the centre is defined in terms of the eccentricity.

Definition 2.1. The *centre* of a graph G is the set of vertices with the minimum eccentricity. In other words, x is in the centre when $\forall v \in G : \mathrm{ecc}(x) \leq \mathrm{ecc}(v)$. Equivalently, the centre is the set $\{v \in G : \mathrm{ecc}(v) = \mathrm{rad}(G)\}$.

Viewing a graph as a communication network, the eccentricity is the time to send a broadcast message from a given vertex to all the other vertices, and the centre is the places in the network from which a broadcast message can reach every vertex in the shortest time.

The eccentricity should not be confused with the different concept called the 'broadcast-time' [39]. The eccentricity allows the message to branch out without limit, so it takes only one time-step to send a message from one vertex to all its neighbours. In contrast, the broadcast-time does not allow branching, and specifies that a vertex can send a message to only one neighbour at a time. Thus it takes $\deg(v)$ time-steps to send a message from the vertex v to all its neighbours.

Returning to the eccentricity, we make a first observation that the eccentricities never have big jumps along a path in any graph.

Proposition 2.2. *For two vertices v_1 and v_2 in a graph, $|\mathrm{ecc}(v_1) - \mathrm{ecc}(v_2)| \leq d(v_1, v_2)$.*

Proof. Without loss of generality, we assume $\mathrm{ecc}(v_1) \geq \mathrm{ecc}(v_2)$, and then try to bound $\mathrm{ecc}(v_1) - \mathrm{ecc}(v_2)$ from above. Also, let $\mathrm{ecc}(v_1) = d(v_1, v_1')$ and $\mathrm{ecc}(v_2) = d(v_2, v_2')$. Now,

$$\begin{aligned}
\mathrm{ecc}(v_1) - \mathrm{ecc}(v_2) &= d(v_1, v_1') - d(v_2, v_2') \\
&\leq d(v_1, v_1') - d(v_2, v_1') && \text{(because } v_2' \text{ is an ecc-wit of } v_2\text{)} \\
&\leq d(v_1, v_2) && \text{(triangle inequality)}
\end{aligned}$$

□

In a general graph, the eccentricity of a vertex is computed by breadth-first search, which runs in time $O(n + m)$. Then, computing the centre requires computing the eccentricities of all vertices, so the overall time-complexity of this brute-force procedure is $O(nm)$. The number of edges m in a graph is no more than $n(n-1)/2$, so $O(nm)$ can also be re-written as $O(n^3)$.

Nevertheless, the centre can be computed more efficiently in specific classes of graphs, which will be discussed in Section 3.2 as well as in Chapter 4.

Median. Like the eccentricity and the centre, another similar pair of concepts is the distance-sum and the median. As opposed to the eccentricity, the distance-sum takes branching into account. A broadcast message in a communication network automatically branches out, but a physical vehicle cannot. Hence, the median models the suitable places to station such service vehicles, such as ambulances.

Definition 2.3. The *distance-sum* of a vertex v, denoted $\mathrm{ds}(v)$, is defined as $\sum_{x \in G} d(v, x)$. Then the *median* of a graph is the set of vertices with the minimum distance-sum.

2.5 Graph Classes

There are several classes of graphs that occur often in this thesis. A *clique* means a *complete graph*, in which every two vertices are adjacent; the clique on n vertices is called an *n-clique*. A *path* (or a *path-graph*, denoted P_n) is a graph with vertices $\{v_1, v_2, \ldots, v_n\}$ and edges $\{v_i, v_{i+1}\}$ for all $i \in [n-1]$. Then in what follows, we survey some basics properties of trees and k-trees, and also briefly discuss chordal graphs. More details of many more graph classes can be found in the dedicated reference book by Brandstädt *et al.* [20].

2.5.1 Trees and k-trees

A *tree* is a graph without cycles. The absence of cycles gives rise to many elegant properties. Firstly, a tree can be defined in the following recursive way.

(T1) The singleton vertex forms a tree.

(T2) Given a tree T, we can build a new tree T' by appending a new vertex to the existing tree T.

In a tree, a vertex of degree 1 is called a *leaf*. (In the case of (T1), the singleton vertex is also considered a leaf despite having degree zero.) The algebraic elegance of such a recursive definition enables us to establish more properties. For example, the number of edges in

a tree is always $n - 1$, and every two vertices are connected via a unique shortest path. Furthermore, every two vertices in a tree are connected by a unique path.

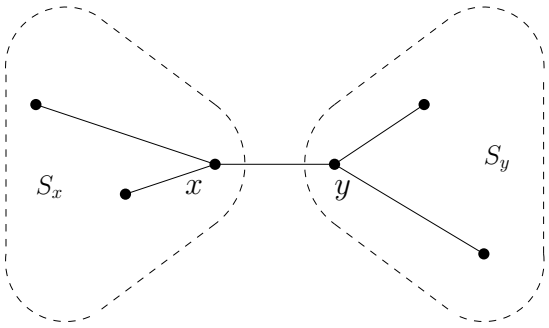


Figure 2.2: Subtrees with respect to an edge.

Definition 2.4. Let x and y be adjacent vertices in a tree. Then the *subtree of x with respect to this edge*, denoted S_x, is the subgraph induced by all the vertices whose paths to y pass through x. Symmetrically, the subtree of y, denoted S_y, is the subgraph induced by all the vertices whose paths to x pass through y. (See Figure 2.2.)

Rooted trees. A *rooted tree* is a tree T with a distinguished vertex r called the *root*. It is convenient to draw a rooted tree with a sense of direction: usually, the root is drawn on the top, and the other vertices are drawn beneath. (See Figure 4.8.)

For every vertex x, the *subtree* of x, denoted S_x, is the tree induced by the set of vertices, whose paths to the root pass through x. Consequently $x \in S_x$. Meanwhile, the *children* of x, denoted $\mathrm{Ch}(x)$, are the vertices in S_x that are adjacent to x. For example, in Figure 4.8, the subtree of v_2 is the tree induced by $\{v_2, v_5, v_6\}$, whereas the children of v_2 are v_5 and v_6.

Definition 2.5. For every vertex v in a rooted T with root r, the *level* of v is defined to be $\mathrm{lev}(v) = d(v, r)$.

We say that a vertex v_1 is on a *deeper* level than another vertex v_2 when $d(r, v_1) > d(r, v_2)$. Meanwhile, from Definition 2.5 we can make the simple observation below.

Proposition 2.6. *For every two adjacent vertices $v_1 \sim v_2$ in a rooted tree,*

$$|\mathrm{lev}(v_1) - \mathrm{lev}(v_2)| = 1.$$

Proof. Without loss of generality, assume that $\mathrm{lev}(v_1) \leq \mathrm{lev}(v_2)$, which is equivalent to $d(v_1, r) \leq d(v_2, r)$. Since $v_1 \sim v_2$ and we are in a tree, the shortest path between v_2 and r must pass through v_1. Therefore

$$\begin{aligned} \mathrm{lev}(v_2) &= d(v_2, r) \\ &= d(v_2, v_1) + d(v_1, r) \\ &= 1 + \mathrm{lev}(v_1), \end{aligned}$$

which establishes $\mathrm{lev}(v_2) - \mathrm{lev}(v_1) = 1$ where $\mathrm{lev}(v_1) \leq \mathrm{lev}(v_2)$, and hence completes the proof. □

With the level-function established, we then present some related concepts that will come to use in Chapter 5.

Definition 2.7. In a rooted tree, a path $[v_1 \dots v_k]$ is *monotone* when either:

- $\mathrm{lev}(v_i) < \mathrm{lev}(v_{i+1})$ for all $1 \leq i < k$, or
- $\mathrm{lev}(v_i) > \mathrm{lev}(v_{i+1})$ for all $1 \leq i < k$.

Note that the path between any vertex and the root is always monotone, and that every segment of a monotone path is also monotone. Furthermore, if a path is not monotone, then it must consist of exactly two monotone segments as shown by Proposition 2.8 below.

Proposition 2.8. *In a rooted tree, a path is either monotone, or can be decomposed into two monotones paths.*

Proof. Consider the path between any two vertices v_1 and v_2. If this path is already monotone, then the statement already holds.

On the other hand, suppose this path is not monotone. Then consider the following two paths $[r \dots v_1]$ and $[r \dots v_2]$. Let u be the last common vertex of these two paths. Then $[v_1 \dots u \dots v_2]$ is the unique path between v_1 and v_2, and indeed this path can be decomposed into two monotones paths. □

The vertex u in the proof above is called a turning-point. Formally, the turning-point of a non-monotone path $[v_1 \dots v_k]$ is the vertex v_i (with $2 \leq i \leq k-1$) such that $\mathrm{lev}(v_{i-1}) > \mathrm{lev}(v_i)$ and $\mathrm{lev}(v_i) < \mathrm{lev}(v_{i+1})$. Hence, Proposition 2.8 can also be phrased as 'every path in a rooted tree has at most one turning-point'.

k-trees. In the recursive definition, (T1) starts with *one* vertex, and (T2) involves adding *one* extra vertex. This can be generalised from *one* to any natural number k, leading to the notion of the k-tree.

Definition 2.9 (k-tree)**.** For $k \in \mathbb{N}$, a *k-tree* is either of the following two:

(kT1) a k-clique, or

(kT2) the graph obtained by joining a new vertex to a k-clique in an existing k-tree.

In this way, a tree is a 1-tree. Similar with the usual trees, a vertex of degree k of $k-1$ in a k-tree is called a *leaf*. Figure 2.3 lists the 1-trees, 2-trees and 3-trees on small numbers of vertices.

A k-tree consists of the k vertices in the starting clique, plus $n-k$ later additions. The starting clique contains $\binom{k}{2}$ edges, and each later addition introduces k edges, so the number of edges can be precisely expressed by the following proposition.

Proposition 2.10. *In a k-tree with n vertices, the number of edges is exactly*

$$\binom{k}{2} + k(n-k).$$

□

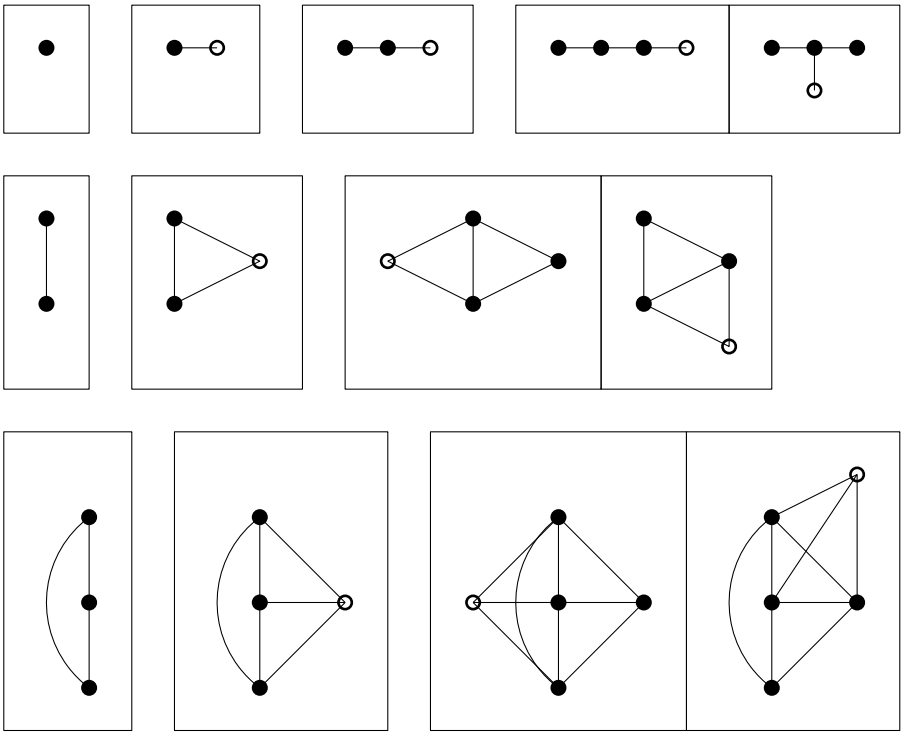


Figure 2.3: The 1-trees, 2-trees and 3-trees on small numbers of vertices.

Every later addition introduces a new subgraph that induces a $(k+1)$-clique, so we can count the number of subgraphs that induce $(k+1)$-cliques. In addition, every $(k+1)$-clique contains exactly $k+1$ subgraphs that induce k-cliques, we can derive the number of subgraphs in a k-tree that induce k-cliques. These are stated in the next proposition.

Proposition 2.11. *In a k-tree with n vertices, there are $n-k$ subgraphs that induce $(k+1)$-cliques, and there are $(k+1)(n-k)$ subgraphs that induce k-cliques.* □

Recall the definitions of the eccentricity and ecc-wits at the start of Section 2.2. Below is a lemma that relates ecc-wits and leaves in k-trees.

Lemma 2.12. *Every vertex in a k-tree has at least one ecc-wit that is a leaf.*

Proof. The base case is the k-clique, in which every vertex has degree $k-1$ and thus is a leaf. For every vertex in the k-clique, the set of ecc-wits consists of all the other vertices, which are all leaves. Hence the statement holds.

Next, assume that a k-tree G satisfies the statement, and consider adding an extra vertex to G. For every existing vertex x, this addition increases $\mathrm{ecc}(x)$ by either 0 or 1. In the first case, one ecc-wit of x is a leaf by the assumption. If $\mathrm{ecc}(x)$ is increased by 0, then this leaf is still an ecc-wit of x. In the second case, if $\mathrm{ecc}(x)$ is increased by 1, then this newly added vertex must have become an ecc-wit of x. This new vertex is a leaf too, so it becomes the desired leaf ecc-wit of x. □

Despite the ecc-wits of a vertex must include a leaf, the converse does not hold. In Figure 2.4, the leaf v is not an ecc-wit of any vertex.

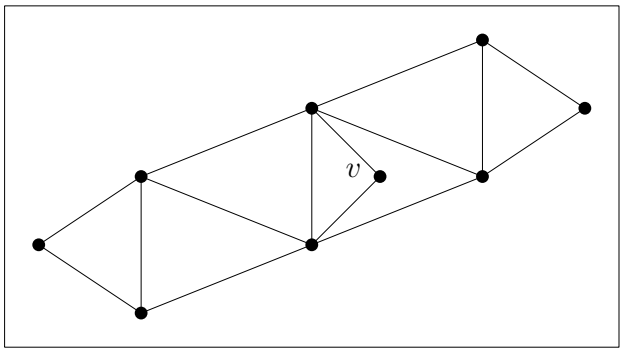


Figure 2.4: A 2-tree where the vertex v is a leaf but not an ecc-wit of any vertex.

2.5.2 Chordal graphs

Chordal graphs are an important class of graphs, and possess many useful properties. In fact, the centres and medians of chordal graphs have been widely studied in the literature, as discussed in Section 3.2.

Definition 2.13. A *chordal graph* is a graph with no induced cycle of length greater than three.

Equivalently, let $[v_1, v_2, v_3, \ldots, v_p]$ be a cycle of length at least four in a chordal graph. This means that $v_i \sim v_{i+1}$ for every $i \in [p-1]$, and $v_p \sim v_1$. Then the chordal property requires that there must be an edge between some v_i with some vertex v_j that is neither v_{i-1} nor v_{i+1}.

In practice, we use the latter criterion more often. Also, note that graphs on fewer than four vertices trivially satisfies the chordal criterion.

Proposition 2.14. *Every complete graph is chordal.*

Proof. Let $[v_1, v_2, v_3, \ldots, v_p]$ be a cycle in a complete graph. But $v_1 \sim v_3$ as the graph is complete. This means that v_1 is adjacent to not just v_p and v_2, and hence satisfies the chordal criterion. □

Proposition 2.15. *Every k-tree is chordal.*

Proof. By the previous proposition, the k-clique is chordal, so the base case is covered. Then consider adding a new vertex v to the k-clique H in an existing k-tree G. The inductive hypothesis assumes that G is chordal. Hence, we need to check if adding v leads to any new induced cycle of length at least four.

Let $[v, u_1, u_2, \ldots, u_p]$ be a cycle of length at least four in G after adding v. Being a cycle, $v \sim u_1$ and $v \sim u_p$. But the neighbours of v form a clique, so $u_1 \sim u_p$, which means that the resultant graph is indeed chordal. □

Despite every k-tree is chordal, not every chordal graph is a k-tree. For example, the chordal graph in Figure 2.5 is not a k-tree for any $k \in \mathbb{N}$. Having only five vertices, one can easily conclude that this graph is chordal by checking if every cycle of length at least four contains a chord. Now, noting the number of vertices $n = 5$ and the number of edges $m = 8$, one can employ Proposition 2.10 to conclude that this graph is not a k-tree for any k. Firstly, this graph is not a 1-tree because it has cycles. Then, a 2-tree on five vertices must have exactly $\binom{2}{2} + 2 \cdot 3 = 7$ edges, so this graph cannot a 2-tree, and this argument can be used for larger values of k.

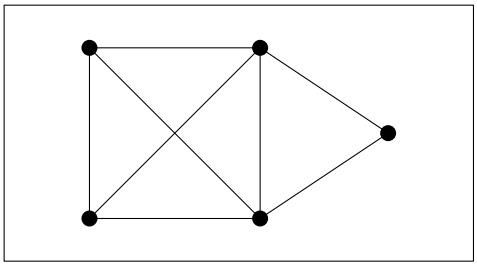


Figure 2.5: A chordal graph that is not a k-tree for any $k \in \mathbb{N}$.

2.6 Integer Compositions

Definition 2.16. A *composition* of $n \in \mathbb{N}$ is a sequence of natural numbers that sum to n. Each number in the sequence is called a *part*. The number of parts in a composition is the *length* of the composition.

To avoid ambiguity, we write compositions in `type-writer font`. For example, $n = 4$ has eight compositions: `1111`, `112`, `121`, `211`, `13`, `31`, `22`, `4`.

Proposition 2.17. *The total number of compositions of n is 2^{n-1}.*

Proof. This proposition can be proven by visualising a natural number n as n balls arranged in a line. Then a composition corresponds to placing separators into the $n - 1$ possible slots between the balls. Now that there are $n - 1$ slots, each either has or does not have a separator, so there are 2^{n-1} ways to place separators. □

We sometimes focus on compositions with a fixed length. For example, there are three compositions of $n = 4$ with exactly three parts: `112`, `121` and `211`.

Proposition 2.18. *The number of compositions of n with k parts $\binom{n-1}{k-1}$.*

Proof. Same as the previous proposition, a composition of n with k parts can be viewed as placing $k - 1$ separators into $n - 1$ slots, and there are $\binom{n-1}{k-1}$ possibilities. □

We can also restrict the possible sizes of the parts. Given a subset S of natural numbers, an S-composition of n is a composition of n whose parts are all in S. For example, there are

Table 2.1: All five $\{1, 2\}$-compositions of 4.

1	111
1	12
1	21

2	11
2	2

five $\{1, 2\}$-compositions of 4: `1111`, `112`, `121`, `211` and `22`. The other three compositions `13`, `31` and `4` are not $\{1, 2\}$-compositions because 3 and 4 are not in $\{1, 2\}$.

Although the number of general S-compositions of n does not have a formula as nice as the numbers of unrestricted and fixed-length compositions, we can still systematically derive the recurrence for the number of S-compositions of n for any given S.

Proposition 2.19. *Let $S \subset \mathbb{N}$, and let $c_S(n)$ denote the number of S-compositions of n. Then for all $n > \max\{S\}$:*

$$c_S(n) = \sum_{x \in S} c_S(n - x).$$

Meanwhile, the bases cases for $n \leq \max\{S\}$ are calculated explicitly.

For example, suppose $S = \{1, 2\}$. Then `1` is the only $\{1, 2\}$-composition of 1, and `11` and `2` are the only $\{1, 2\}$-compositions of 2. Hence, the base cases are $c_S(1) = 1$ and $c_S(2) = 2$. Next, every $\{1, 2\}$-composition of 4 can start with either 1 or 2, and the tail of every $\{1, 2\}$-composition of 4 that starts with `1` is a $\{1, 2\}$-composition of 3, while the tail of every $\{1, 2\}$-composition of 4 that starts with `2` is a $\{1, 2\}$-composition of 2. Hence, for all $n > 2$, $c_S(n) = c_S(n - 1) + c_S(n - 2)$.

This recurrence allows us to list the S-compositions systematically. Table 2.1 lists the five $\{1, 2\}$-compositions of four. In the left-hand table are the three $\{1, 2\}$-compositions of four that start with `1`, which are obtained by prepending a `1` to every $\{1, 2\}$-composition of three. In the right-hand table are the two $\{1, 2\}$-compositions of four that start with `2`, which are obtained by prepending a `2` to every $\{1, 2\}$-composition of two.

Chapter 3

Background

O sprich mir nicht von jener bunten Menge,
Bei deren Anblick uns der Geist entflieht.
Verhülle mir das wogende Gedränge,
Das wider Willen uns zum Strudel zieht.

—Goethe (Faust I. Prelude in the theatre) [1]

This chapter's three sections provide brief surveys of the three separate fields related to this thesis—one in algorithms and data structures, one in graph theory, and one in combinatorics.

The first background area is Graph Preprocessing, which is related to the simplification method *partition-graphs* presented in Chapter 5. Graph Preprocessing is in turn related to topics such as *graph spanners*, *graph sparsifiers*, *distance oracles* and *graph clustering*, which are useful tools in the popular fields like Graph Databases [7] and Network Science [83]. Section 3.1 outlines the main graph preprocessing methods that are relevant to this thesis.

In order to gain insight on the original graph from the simplified graph, some properties

[1] Oh don't speak to me of that mixed crowd, / At whose sight the mind escapes. / Veil from me the surging rush, / Which pulls us into the whirlpool against our will.

of the original graph need to be preserved in some well-defined way by the simplified graph. The preprocessing method presented in Chapter 5 turns out to preserve graph centrality for trees—more specifically the *centre* and the *median*. Chapter 4 gathers some basic results on the centre and median, which are then applied in Chapter 5. The centre and median, and graph centrality in general, are part of a considerable line of research, and these existing works are surveyed in Section 3.2.

The preprocessing method *partition-graphs* in Chapter 5 does not produce a unique outcome on a given input graph, so it is of interest to study the set of all possible partition-graphs on a fixed input graph. Chapter 6 initiates this direction by focusing on paths. Our preprocessing method is based on grouping vertices, and consequently an outcome on the n-vertex path P_n can be viewed as a sequence of numbers that sum to n. Such a sequence is an *integer composition* with restricted part-sizes, which is a significant subfield in combinatorics. Section 3.3 outlines the relevant results about the enumeration and pattern-avoidance of integer compositions.

3.1 Graph Preprocessing

The general umbrella term *graph preprocessing* refers to techniques that turn a large input graph into a more succinct representation, so that we can perform more efficient computations on the succinct representation to gain insight on the input graph. As many graphs in applications are so large that some computations become infeasible, preprocessing is a vital means to handle large-scale graphs. The main graph preprocessing tools include *graph spanners*, *graph sparsification*, *distance oracles* and *graph clustering*, among others. These are discussed one by one in the rest of this section.

For any function $f(n)$, the notation $\tilde{O}(f(n))$ means $O(f(n) \cdot \log^k n)$ for some $k \in \mathbb{N}$.

3.1.1 Graph spanners

The number of edges in a graph affects the running time of many algorithms, so it is desirable to reduce the number of edges in a graph but still approximately preserving the distance function. This is the motivation behind the subject of *graph spanners*, on which we give a brief outline in this section. With research effort spanning the past three decades, graph spanners have grown into a substantial field, clocking over 150 references in the recent thirty-page review article by Ahmed *et al.* [2].

In the beginning, a *spanner* of a graph was defined to be a spanning subgraph that approximately preserves the distance function of the original graph. Specifically, given a connected graph G and a parameter $t \in \mathbb{N}$ called the *stretch*, a *t-spanner* H is a spanning subgraph of G such that for every pair of vertices v_1 and v_2 in $V(G) = V(H)$, the distance $d_H(v_1, v_2)$ is never bigger than t times the original distance $d_G(v_1, v_2)$; that is,

$$d_H(v_1, v_2) \leq t \cdot d_G(v_1, v_2). \tag{3.1}$$

In order to satisfy this inequality, $d_H(v_1, v_2)$ has to be well-defined for every pair v_1 and v_2, which implicitly means that a spanner must be connected. Meanwhile, as edges are removed, the shortest path between any two vertices cannot become shorter. Hence, there is an implicit lower bound on $d_H(v_1, v_2)$, and the overall inequality is the following:

$$d_G(v_1, v_2) \leq d_H(v_1, v_2) \leq t \cdot d_G(v_1, v_2).$$

The process of constructing spanners should be efficient. In addition, the challenges of constructing effective spanners are removing as many edges from G as possible and keeping the parameter t as small as possible. This leads to the *minimum t-spanner* problem, which takes a graph G, a parameter t and an integer m' as inputs, and decides if G has a t-spanner with at most m' edges. As Peleg and Schäffer [94] showed in one of the earliest works on

spanners, the minimum t-spanner problem is NP-complete. Still, the same article proved that for every n-vertex graph and every number $k \in [n]$, there exists a $(4 \log_k n + 1)$-spanner with at most kn edges, which can be constructed in polynomial time.

As the minimum t-spanner is computationally difficult, a series of research studied approximation algorithms for this problem. It turned out that approximating the minimum t-spanner is also computationally difficult. In this series' first work, Kortsarz [71] showed that for every $t \in \mathbb{N}$, approximating the minimum t-spanner is at least as hard as approximating the set cover problem. Later, Elkin and Peleg [38] surveyed the hardness of approximability for a number of variants of the minimum t-spanner problem.

Meanwhile, some recent effort attempted to devise efficient algorithms to construct near-optimal t-spanners with a given bound on t. For example, Tatikonda *et al.* [115] presented a quadratic-time algorithm to construct spanners, while Veluri, Jayakumar and Nair [119] characterised an infinite class of graphs, on which the minimum t-spanner problem can be solved in linear time.

The original definition of spanners only has one multiplicative parameter, so subsequent research extended the original concept with an additional additive parameter. For two parameters $\alpha \geq 1$ and $\beta \geq 0$, a spanner H is called an (α, β)-*spanner* when it satisfies

$$d_G(v_1, v_2) \leq d_H(v_1, v_2) \leq \alpha \cdot d_G(v_1, v_2) + \beta. \tag{3.2}$$

Hence, the t-spanner defined earlier is simply a $(t, 0)$-spanner.

Elkin and Peleg [37] showed that for any constants $\epsilon, \lambda > 0$, there exists a function $\beta = \beta(\epsilon, \lambda)$ such that a $(1 + \epsilon, \beta)$-spanner of size $O(n^{1+\lambda})$ can be constructed efficiently. Meanwhile, Baswana *et al.* [9] presented a linear-time algorithm to construct a $(k, k-1)$-spanner with size $O(kn^{1+1/k})$ for any $k \in \mathbb{N}$.

One can also consider spanners with only an additive parameter. Such a $(1, \beta)$-spanner is called an *additive spanner* when β is a function, and is called a *purely additive spanner*

when β is a constant. On purely additive spanners, the following authors showed how to construct $(1, \beta)$-spanners with their respective sizes:

- 1999: (1,2)-spanners of size $O(n^{3/2})$ by Aingworth *et al.* [4]
- 2010: (1,6)-spanner of size $O(n^{4/3})$ by Baswana *et al.* [9]
- 2013: (1,4)-spanner of size $\tilde{O}(n^{7/5})$ by Chechik [25]

When β is a function, Chechik [25] presented a construction for additive spanners with $\beta = \tilde{O}(\sqrt{n^{1-3\delta}})$ and size $\tilde{O}(n^{1+\delta})$, for any $3/17 \leq \delta < 1/3$.

There is also a generalisation called *pairwise spanners*. In this context, the original definition of spanners can be considered *all-pair* spanners, where the distance of every pair of vertices is preserved, and Inequality (3.1) is satisfied for every pair in $V \times V$. In a pairwise spanner, however, Inequality (3.1) only needs to hold for a given subset of pairs $P \subseteq V \times V$. Below are some examples of work in this line of research, with P being any arbitrary subset of $V \times V$:

- 2013: Pairwise $(1, 4 \log n)$-spanner of size $O(n \cdot (|P| \log n)^{1/4})$
  by Cygan, Grandoni and Kavitha [33]
- 2013: Pairwise (1,2)-spanners of size $\tilde{O}(n \cdot |P|^{1/3})$ by Kavitha and Varma [67]
- 2017: Sparse pairwise (1,4)-spanners and (1,6)-spanners by Kavitha [66]

There are many more variants of spanners. So far we only focused on spanners of unweighted and undirected graphs, but there are works studying spanners of weighted graphs and directed graphs as well. Just a few years after Peleg and Schäffer's initial work on spanners for unweighted graphs, Althöfer *et al.* [6] presented a polynomial-time algorithm to construct sparse spanners for weighted graphs.

On directed graphs with or without edge-weights, Berman *et al.* [14] studied approximation problems for spanners. Namely, they presented an $O(\sqrt{n} \log n)$-approximation algorithm

to compute sparse spanners on weighted directed graphs, and established an $O(n^{1/3} \log n)$-approximation algorithm to compute 3-spanners on directed unweighted graphs.

In the unweighted setting, Chlamatáč *et al.* [29] studied a number of approximation problems for different variants of spanners; among the paper's many results, they developed a polynomial-time $O(n^{2/5+\varepsilon})$-approximation algorithm to compute pairwise spanners for both undirected and directed graphs.

3.1.2 Sparsification

As the etymology suggests, the term *sparsification* denotes techniques for constructing a sparse graph to approximately represent the original graph. The sparser graph is called a *sparsifier*, and in order for it to represent the original graph, the sparsifier needs to approximately preserve certain properties of the original graph.

A *cut sparsifier* is designed to preserve cut-related properties. A *cut* of a connected graph G is a set of edges C, such that G becomes disconnected when the edges in C are removed. Also, the *weight* of a cut C in an edge-weighted graph is the sum of the weights of the edges in C, and a minimum cut is one that has the minimum weight over all cuts of G. Given a graph G and an error parameter $\varepsilon \geq 0$, Benczúr and Karger [12, 13] established the existence of a sparsifier H with $O(n \log n/\varepsilon^2)$ edges, such that the weight of every cut in H is within $1 + \varepsilon$ times the weight of the corresponding cut in G. Furthermore, these authors also showed that H can be constructed in $O(m \log^2 n)$ times if G is unweighted, and in $O(m \log^3 n)$ times if G is weighted.

A *spectral sparsifier*, stronger than a cut sparsifier, aims to preserve the original graph's Laplacian spectrum, which in turn captures connectivity-related properties. With $A(G)$ denoting the adjacency matrix of G and $D(G)$ denoting the diagonal matrix that lists the degrees of G's vertices, the *Laplacian matrix* $L(G)$ is defined to be $D(G) - A(G)$. Then the *Laplacian spectrum* of G is the set of eigenvalues of $L(G)$. The spectra of the Laplacian matrix and other matrices form the subject of Spectral Graph Theory [32].

Formally, for any undirected and weighted graph G, a $(1+\varepsilon)$-*spectral sparsifier* is a sparse graph H that satisfies

$$(1-\varepsilon)\cdot \mathbf{x}^T L(H)\mathbf{x} \quad \leq \quad \mathbf{x}^T L(G)\mathbf{x} \quad \leq \quad (1+\varepsilon)\cdot \mathbf{x}^T L(H)\mathbf{x}, \tag{3.3}$$

for all n-dimensional real-valued vectors $\mathbf{x}$. Under this framework, a cut sparsifier also satisfies Inequality (3.3) when $\mathbf{x}$ is an n-dimensional binary vector.

In the first work on spectral sparsification, Spielman and Teng [112] presented an algorithm that runs in $O(m \log^c m)$ time (with c being an absolute constant), and uses random sampling to construct a spectral sparsifier of nearly-linear size (number of edges).

This was soon improved by Spielman and Srivastava [111], who presented a randomised algorithm that runs in nearly-linear time, and constructs a spectral sparsifier with $O(n \log n/\varepsilon^2)$ edges. A key in this algorithm is a nearly-linear time procedure that builds a data structure which returns the approximate *resistance* between any two vertices in $O(\log n)$ time, where the resistance is a function on pairs of vertices, somewhat analogous to the distance function.

Later, Batson, Spielman and Srivastava [11] developed a deterministic $O(n^3 m)$-time algorithm that constructs a spectral sparsifier with $O(n/\varepsilon^2)$ edges. Allen-Zhu, Liao and Orecchia [5] then improved the running time of this result by leveraging a connection between sparsification and a regret minimisation problem over density matrices, and obtained an $O(n^{2+\varepsilon})$-time algorithm. Most recently, Lee and Sun [76] combined the techniques of random sampling from [111] and barrier functions from [5, 11] to achieve an algorithm that constructs a sparsifier with $O(n/\varepsilon^2)$ edges in almost-linear time. The following list summarises the history outline above.

- 2011: Time $\tilde{O}(m)$. Size $\tilde{O}(n/\varepsilon^2)$ edges. Spielman and Teng [112]
- 2011: Time $\tilde{O}(m)$. Size $O(n \log n/\varepsilon^2)$ edges. Spielman and Srivastava [111]
- 2012: Time $O(n^3 m)$. Size $O(n/\varepsilon^2)$ edges. Batson, Spielman and Srivastava [11]

- 2015: Time quadratic $O(n^{2+\varepsilon})$. Size $O(n/\varepsilon^2)$ edges. Allen-Zhu, Liao and Orecchia [5]
- 2018: Time almost linear. Size $O(n/\varepsilon^2)$ edges. Lee and Sun [76]

3.1.3 Distance oracles

A *distance oracle* of a graph G is a data structure H that efficiently answers distance queries. Unlike sparsifiers and spanners, a distance oracle is not itself a graph but a more intricate object. But similar to spanners, a distance oracle has a *stretch* factor $t \in \mathbb{N}$, and upon a distance query $d_G(v_1, v_2)$ it returns an answer $d_H(v_1, v_2)$ that satisfies

$$d_G(v_1, v_2) \leq d_H(v_1, v_2) \leq t \cdot d_G(v_1, v_2).$$

Apart from the stretch, the other three challenges with distance oracles are the preprocessing time, the query efficiency and the size of the data structure.

The concept of distance oracles was first introduced by Thorup and Zwick [117], who developed a randomised algorithm to construct distance oracles for undirected weighted graphs. For every $k \in \mathbb{N}$, their algorithm takes an expected running time of $O(kmn^{1/k})$ to construct a distance oracle of size $O(kn^{1+1/k})$, and this distance oracle can answer distance queries in $O(k)$ time with stretch $2k - 1$.

Baswana and Sen [10] improved this preprocessing time to $O(n^2)$ on unweighted graph, and then Baswana and Kavitha [8] established that a preprocessing time of $O(n^2)$ is possible on weighted graphs. Wulff-Nilsen [122, 123] improved the preprocessing time and the query time even further, with $O(\log k)$ being the improved query time.

Striving for constant query time, Mendel and Naor [84] presented the construction of distance oracles with processing time $O(n^{2+1/k} \log n)$, size $O(n^{1+1/k})$ and constant query time, with a stretch of $O(k)$. On the other hand, Chechik [26] developed distance oracles of the same size and stretch as Thorup and Zwick's, and achieved constant query time at the cost of a higher preprocessing time.

Another research direction studies the trade-off between the stretch and the size of distance oracles, and it is based on the widely believed and partially proven conjecture: for every $k \in \mathbb{N}$, there exist graphs with $\Omega(n^{1+1/k})$ edges and shortest cycle of length $2k+2$. Thorup and Zwick [117] also showed that in order to achieve a stretch smaller than $2k-1$, any distance oracle needs to occupy at least $\Omega(n^{1+1/k})$ space. On sparse graphs, works that studied these lower bounds include Sommer, Verbin and Yu [110] and Pătraşcu and Roditty [96].

There is a substantial body of work on distance oracles for *planar* graphs with stretch $1+\varepsilon$. Initially, Thorup [116] presented such distance oracles for weighted planar graphs with $O((1/\varepsilon)^2 n \log^3 n)$ preprocessing time, $O((1/\varepsilon) n \log n)$ space and $O(1/\varepsilon)$ query time. The trade-off between the size, the preprocessing time and the query time was then successively improved by a number of papers, with the most recent ones by Gu and Xu [53], Chan and Skrepetos [22] and Charalampopoulos *et al.* [24]. The latest paper by Charalampopoulos *et al.* [24] showed how to construct a distance oracle for a planar graph in roughly $O(n^{3/2})$ preprocessing time, with any of the following three trade-offs:

- Size $\tilde{O}(n^{1+\varepsilon})$. Query time $O(\log^{1/\varepsilon} n)$ for $0 < \varepsilon \leq 1/2$.
- Size $O(n \log^{2+1/\varepsilon} n)$. Query time $O(n^{2\varepsilon})$ for $0 < \varepsilon$.
- Size $n^{1+o(1)}$. Query time $n^{o(1)}$.

3.1.4 Graph clustering

Often in applications, certain subsets of a graph's vertices appear denser, while very few edges connect these dense parts. These dense subgraphs are called *clusters* or *communities*, and they can be viewed as groups of vertices that are closely connected. Given a large-scale graph, it is desirable to find the clusters; a more restricted version of this problem is to partition the vertices into a given number of clusters of prescribed sizes, while a more general problem is to discover the community structure of a large social network. This topic comes under the names

graph clustering [64, 102], *graph partitioning* [18] and *community detection* [44, 45], and has attracted much attention lately. Graph Clustering bears resemblance to our simplification method presented in Chapter 5, although the objectives are different.

Theoretically, graph clustering can be framed as an optimisation problem of partitioning the vertices into k clusters, while minimising the number of edges between distinct clusters. This problem is NP-hard, so in order to handle such a computationally difficult problem, one often resorts to heuristics and approximation algorithms. This has become a vibrant field of research, with countless methods and applications—Fortunato's 2010 survey article [44] alone contains over 450 references! Surveys of the field of graph partitioning include the book edited by Bichot and Siarry [18], and the review articles by Fortunato [44, 45] and Schaeffer [102].

Many of the graph partitioning techniques are based on *repeated bisection*, where the graph is first split into two near-optimal clusters of equal size using some algorithm, and the algorithm is then applied to each cluster. One of the earliest and prominent technique, motivated by the design of electrical circuits, is the heuristic algorithm developed by Kernighan and Lin [68]. The Kernighan-Lin algorithm starts with an arbitrary bisection, and repeatedly improves the bisection by finding a pair of vertices from each cluster to swap. Each iteration of the improvement stage runs in time $O(n^2 \log n)$, so this algorithm is considered efficient for a constant number of swaps.

Another popular bisection technique is *spectral bisection* [64], which uses the relationship between the Laplacian spectrum and the weight of cuts. Based on the theory developed by Fiedler [42], an optimal bisection of a graph can be obtained from the eigenvector associated with the second-smallest eigenvalue of the Laplacian matrix.

The original notion of partitioning assumes the clusters to be disjoint. That is, no vertex can belong to two clusters at the same time. However, motivated by applications, sometimes it is useful to have overlapping clusters too. Examples of works that studied this extension include Mishra *et al.* [87, 88] and Whang, Gleich and Dhillon [120].

3.1.5 Others

We end this section by mentioning a few more topics related to graph preprocessing.

Edge-contraction. Recall that in the context of spanners, edge-removal always increases the distance and leads to the inequality

$$d_G(v_1, v_2) \leq d_H(v_1, v_2) \leq \alpha \cdot d_G(v_1, v_2) + \beta.$$

On the other hand, edge-contraction is another operation that can also serve as the basis of graph simplification.[2] However, contrary to edge-removal, edge-contraction always decreases the distance, so Inequality 3.4 below naturally becomes the distance-approximation property in the context of edge-contraction.

$$\frac{1}{\alpha} \cdot d_G(v_1, v_2) - \beta \leq d_H(v_1, v_2) \leq d_G(v_1, v_2). \tag{3.4}$$

Given a graph G and constants α and β, Bernstein *et al.* [15] studied a number of related optimisation problems of finding a minimal set of edges to contract such that Inequality (3.4) is satisfied. Their results are the computational complexities of different combinations of α and β.

Graph sketches. Apart from building smaller graphs (by removing or contracting edges) or intricate data structures (distance oracles), one can also project a graph to numerical vectors. Projecting a large dataset to a more succinct numerical representation is a well-established idea called *sketches*. This idea can also be applied to graphs. Ahn, Guha and McGregor [3] studied *graph sketches*, which are ways to represent a graph as numerical vectors, while preserving desirable properties such as the size of the cuts, the distance function and the prevalence of dense subgraphs.

[2] Our *partition-graph* construction introduced in Chapter 5 can also be viewed as edge-contractions.

Metric embeddings. The notion of distance-distortion is also related to the field of *metric embeddings*. Originally a purely mathematical topic that studies whether a given metric space can be embedded into other metric spaces, metric embeddings have found their use in computer science too.

Metric embeddings can be used as a technique to approach computational problems that involve metric spaces. Many computational problems involve metric spaces of some sort—the vertices of graph form a metric space with the metric being the shortest-path distance. In fact, metric embeddings were first introduced to Computer Science by Linial, London and Rabinovich [77] who studied embeddings of graphs and used them to solve some flow-related problems. Later, metric embeddings also found uses in computer vision where the metric space consists of images, and in computational biology where the metric space consists of DNA sequences.

The general idea of metric embeddings is to map the original objects into the Euclidean space, such that the distance between any two original objects approximates the distance between their images in the Euclidean space. When constructing metric embeddings, the main goals include small dimension of the target space and small distance distortions.

The typical target metric space is the Euclidean space $\mathbb{R}^k$ with the l_p-metric:

$$d(\mathbf{x}_1, \mathbf{x}_2) = \|\mathbf{x}_1 - \mathbf{x}_2\|_p \qquad \text{and} \qquad \|\mathbf{x}\|_p = \left(\sum_{i=1}^{k} |x_i|^p \right)^{\frac{1}{p}}.$$

With a suitable embedding, one can then gain a new perspective on the original objects, and sometimes an embedding can lead to approximation algorithms for problems that are intractable in the original domain.

As a final note, embeddings are injective and most likely not surjective, while graph simplification is mostly surjective and most likely not injective. Hence the theme of this thesis is distinct from metric embeddings, although they are in common in the aspect of distance-distortion.

The topic of metric embeddings and their applications is comprehensively treated in the long paper by Abraham, Bartal and Neiman [1] and the book by Ostrovskii [92].

Graph compression. Preprocessing can also be viewed from the angle of data compressing, which is also based on grouping vertices, but places more emphasis on how to recreate the original graph. The compression scheme developed by Navlakha, Rastogi and Shrivastava [90] consists of a summary-graph and a correction-set. Each vertex in the summary-graph corresponds to a set of original vertices (which does not have to induce a connected subgraph). This scheme was later extended to weighted graphs by Toivonen *et al.* [118].

The scheme above is a lossless compression, which recreates the exact original graph; a survey of lossless graph compression methods is provided by Besta and Hoefler [16]. On the other hand, if one can accept lossy compression that recreates only an approximation of the original graph, one might be able to compress the graph even more. A practical programming framework for lossy graph compression was developed by Besta *et al.* [17]. Meanwhile, the framework developed by Shin *et al.* [103] supports both lossless and lossy compression.

Graph streaming. When a graph is too large to fit in a computer's memory, only a small part of the graph can be accessed at every given moment. Algorithms with this constraint need to compute properties of the entire graph based only on the local observations through a 'window' that moves across the graph. This is related to well-established concept called the *data stream model*, where the term 'stream' alludes to newer data flowing into the window and flushing older data out. Data streams were originally motivated by the study of network traffic and other application domains, and can also be applied to large-scale graphs. Existing graph algorithms need to be revised under this streaming setting. The article by McGregor [81] provides a survey of the streaming versions of algorithms for spanners, sparsification, matchings and subgraph-counting.

3.2 Graph Centrality

Though seemingly intuitive, the notion of the 'centre' of a graph turns out hard to pinpoint mathematically. As a result, countless notions of centrality have accumulated over the years, some defined in terms of distance and some in terms of flow and connectivity. The sheer number is demonstrated by Reid's survey article [98], which in its abstract lists up to 43 different centrality concepts! Despite this overabundance, most of the centrality concepts are common in the sense that each of them defines a set of vertices that optimise some numerical function called the *centrality measure*. For example, as defined in Section 2.4, the *centre* of a graph is the set of vertices that minimise the eccentricity function, and the *median* of a graph is the set of vertices that minimise the distance-sum function.

The centre and the median are distance-based centrality concepts. On the other hand, the *branch-weight centroid* and the *cutting centre* are centrality concepts not defined in terms of the distance function. The *branch-weight* of a vertex v in a tree T, first introduced by Zelinka [125], is the maximum number of vertices in a connected component of $T \setminus \{v\}$, and then the *branch-weight centroid* is the set of vertices that minimise the branch-weight function.

The *cutting number* of a vertex v in a graph G is defined by Harary [58, 59] to be the number of vertex-pairs u, w of G—with u, w and v all distinct—such that every path between u and w contains v. Then the *cutting centre* of G is the set of vertices with the *maximum* cutting number. The cutting number is a connectivity-based centrality measure. A high cutting number means that the vertex is an important 'bridge' that connects pairs of other vertices. The main connectivity-based centrality known nowadays is the *betweenness centrality* [19], which is closely related to the cutting number and will be mentioned again near the end of this section.

Due to the wide applicability of graphs and the prevalence of the notion of the 'centre' of a graph, a single centrality concept may bear different names that were independently coined by researchers from different fields. For example, in the distant past, the distance-sum

function was called the *status* by Harary [57] in 1959, the *point-centrality* by Sabidussi [100] in 1966, or simply the *distance* by Reid [97] and Slater [104, 105] from the 1970s onwards.

Different centrality concepts may coincide in certain graph classes. For example, the median and the branch-weight centroid were defined differently, but Zelinka [125] showed that these two concepts are equivalent in trees. Probably due to this, Slater once used the term 'centroid' to refer to the median in 1978 [104], but later returned to the more commonly accepted term 'median' in 1980 [105].

This thesis focuses only the distance-related *centre* and *median*. The study of centrality can be roughly categorised into two main strands. For specific classes of graphs, the *descriptive* aspect seeks structural characterisations of the central vertices, while the *algorithmic* aspect seeks efficient algorithms to locate them. Below we summarise the existing works on both the descriptive and algorithmic aspects of the centre and the median, starting from the simplest classes of trees towards the more general k-trees and chordal graphs. At the end of this present section, we mention some other centrality concepts and related problems.

Centres and medians of trees The centre and median of a tree were first studied by Camille Jordan[3] in his 1869 article [63]. In this historical article, Jordan established that both the centre and the branch-weight centroid of a tree each consists of either a single vertex or two adjacent vertices, but also showed that these two sets do not necessarily coincide. Later, Zelinka [125] proved that the median and the branch-weight centroid of a tree are the same set of vertices, so consequently the median of a tree also consists of either a single vertex or two adjacent vertices. This structural characterisation of the centre and the median of a tree was also mentioned by Lovász [78, 6.21a] and by Graham, Entringer and Székely [50].

As for the algorithmic aspect, the centre of a tree can be easily located using a leaf-removing procedure originally presented by Goldman [47, 48, 49]. This is an implicit al-

[3]Camille Jordan (1838–1922) was a French mathematician (thus pronounced with a guttural R and a nasal A) with contributions in group theory and analysis, and hence not to be confused with the German geodesist Wilhelm Jordan (1842–1899) known for Gauss-Jordan elimination, or with the German physicist Pascual Jordan (1902–1980) known for Jordan algebras in quantum mechanics.

gorithm that does not need to compute explicit eccentricity values, with a quadratic runtime. Despite linear-time algorithms were later presented by Handler [56] and Hedetniemi *et al.* [60], this original leaf-removing procedure is still advantageous in its intuitive clarity, and is easier to execute by hand. This will be discussed in Section 4.1 in more detail. As for the algorithmic aspect of the median of a tree, Section 4.2 will present a linear-time algorithm that locates the median of a tree by explicitly computing the distance-sum value of every vertex.

Centres and medians of k-trees There do not seem to be many works exclusively on the centre and median of a k-tree, probably because k-trees belong to the class of chordal graphs, which receive more focus. Nevertheless, Slater [105] showed that the median of a 2-tree induces a 1-clique, 2-clique or 3-clique. In a *maximal outerplanar graph*, a particular type of 2-tree, Proskurowski [95] showed that the centre induces one of seven possible graphs, and Farley and Proskurowski [40] presented a linear-time algorithm to locate the centre. For general k-trees, Granot and Skorin-Kapov [51] presented a linear-time algorithm to locate the centre by explicitly computing the eccentricity of every vertex.

Centres and medians of chordal graphs On the descriptive aspect, Laskar and Shier [74] first showed that the centre of a chordal graph always induces a connected subgraph. Later, Chang [23] established that the centre of a chordal graph induces a biconnected subgraph with diameter at most five. Some further properties of a chordal graph's centre are surveyed in the introduction of the article by Yeh and Chang [124]. As for the algorithmic aspect, Chepoi and Dragan [28] presented an intricate linear-time algorithm that locates the centre of a chordal graph by explicitly computing the eccentricities.

The median of a chordal graph does not exhibit properties as elegant as the centre. For example, the median of a chordal graph does not always induce a connected subgraph; in order for the median subgraph to be connected, Wittenberg [121] showed that a certain neighbourhood condition needs to be satisfied. Meanwhile, Lee and Chang [75] showed that

in order for the median of a chordal graph to induce a clique, the graph needs to be a *strongly chordal* graph—a special type of chordal graph that does not contain a particular induced subgraph.

Centres in other graph classes There are works on the centres in other classes of graphs. The structural properties of the centres of *distance-hereditary graphs* were studied by Yeh and Chang [124]. The centres of *interval graphs* (which are chordal) can be located by a linear-time algorithm by Olariu [91], while the centres of *circular-arc graphs* (which include interval graphs but are not necessarily chordal) can also be located by a linear-time algorithm by Mandal, Pal and Pal [80].

3.2.1 Other centrality variants

Absolute centre If vertices model buildings and edges model roads, then the 'centre' can be viewed as the optimal locations to place certain facilities. Often the facilities have to be in existing buildings, so the 'centre' is required to be vertices. However, sometimes a facility can be placed on any point on a road, so the 'centre' does not always have to be vertices. Thus, one can extend the eccentricity function to cover any *point* in a weighted graph, which is either a vertex or an arbitrary point on an edge. Then an *absolute centre* is any point in the graph that minimises the extended eccentricity. In contrast with the absolute centre, the term *vertex centre* restricts the possible points to vertices, which is the centre as defined in Section 2.4. This antithetical pair of the absolute centre and the vertex centre was first introduced by Hakimi [54] in 1964, and subsequently studied by Halfin [55] in 1974 and by Minieka [86] in 1981. The development of this topic is also summarised by Tansel in his relatively recent 2011 chapter [114].

Parameterised centrality In the study of graph centrality, there are also the parameterised versions of the centre and the median. The original centre-locating problem seeks *individual* vertices that minimise the eccentricity. However, in a real-life graph we often

place multiple facilities and aim to ensure that the facilities *collectively* minimise the eccentricity. More formally, with the eccentricity function extended to vertex-subsets, the *p-centre problem* (with $p \in \mathbb{N}$) seeks a subset S containing p vertices, such that the collective eccentricity $\mathrm{ecc}(S)$ is minimised. The *p-median problem* is defined analogously. An example of an early work is the article by Kariv and Hakimi [65]. The p-centre and the p-median problems form a substantial topic on their own, and being NP-hard, their heuristic algorithms are still actively studied to this day. Examples of recent works include Mladenović *et al.* [89] and Daskin and Maass [34].

One can add different types of parameters to generalise a centrality concept into different directions. For example, Slater [104] studied a generalised version of the centre and the median by restricting the eccentricity and the distance-sum to a given subset of vertices. Given a subset $S \subseteq G$, the generalisations are called the *S-centre* and the *S-median*, which are equivalent to the original centre and median when $S = G$. Also in [104] Slater introduced the *k-centrum* (given $k \in \mathbb{N}$), where the vertices in the k-centrum are required to minimise the following function:

$$r_k(v) = \max\left\{\sum_{x\in S} d(v,x) \mid \text{ for all } k\text{-vertex subsets } S \subseteq G\right\},$$

Thus the 1-centrum is simply the centre, while the n-centrum is the median.

Slater [106] also introduced the *k-nucleus* (with $k \in \mathbb{N}_0$), which is the set of vertices v that minimise the following function:

$$\rho_k(v) = \sum\left\{d\Big(x, B(v,k)\Big) \mid x \in G\right\},$$

where $B(v,k) = \{x \in G \mid d(x,v) \le k\}$ denotes the set of vertices within distance k from v. Consequently, the k-nucleus is equivalent the centre when $k = \mathrm{rad}(G)$, and is equivalent to the median when $k = 0$. Later, Reid [97] studied a generalised version of the branch-weight centroid with a parameter $k \in \mathbb{N}_0$, where the case $k = 0$ is identical to the original centroid.

Reverse problems Most of the works on graph centrality studied the 'centre' of given graphs. For example, given a chordal graph, how can its centre be efficiently located, or what are the properties of its median. On the other hand, there are works that studied the set of graphs with a given 'centre'. Slater [105] showed that for any graph J, there exists a graph whose median subgraph is isomorphic to J. Buckley, Miller and Slater [21] showed that for any graph H, there exists a graph whose centre subgraph is isomorphic to H. This line of research was concluded by Smart and Slater [108] who showed that given any three graphs H, J, K and a natural number $k \geq 4$, there exists a graph whose centre, median and centroid subgraphs are isomorphic to H, J and K respectively, and that the distance between any two of these subgraphs is at least k.

Betweenness and Closeness The popular field of Social Network Analysis [41] uses centrality concepts that are slightly different from the terms used by the graph-theory community. The *betweenness* and the *closeness* metrics are the two main centrality concepts here. The betweenness metric is defined in terms of flow and connectivity, while the closeness metric is essentially the median. In applications, one needs heuristics and preprocessing methods to efficiently compute the different 'centres' of large-scale graphs. For example, Brandes [19] developed an algorithm to compute the betweenness metric that is faster than the direct method, and Sarıyüce *et al.* [101] presented a partitioning method to compute the exact betweenness and closeness metrics. Note that the use of partitioning to compute centrality metrics is another instance of graph preprocessing discussed in the previous Section 3.1.

3.3 Integer Compositions

Chapter 6 takes a closer look on the partition-graphs of paths. Every partition-graph of a path can be viewed as an integer composition with restricted parts, so we can make use of some elegant combinatorial reasoning techniques.

In Chapter 6, we also propose randomised algorithms to generate partition-graphs on paths, and the outcomes of these algorithms turn out to be integer compositions that avoid certain substrings. There is a rich line of research on *pattern-avoiding compositions*, but the 'substrings' in our context are in fact different from 'patterns'. This will be discussed in the first part of this section.

It is a natural goal to investigate the expected number of super-vertices in the outcome of the randomised algorithm proposed in Section 6.2. This question happens to coincide with the *discrete parking problem*, so the second part of the present section provides a historical outline of this well-established topic.

The study of integer compositions is a rich subject with a long history dating back to at least 1893, but here it may suffice to cite only the book by Heubach and Mansour [61] for its good narration of the history and its comprehensive survey of results.

The two central objects studied by this book are *compositions* and *words*. For $S \subset \mathbb{N}$, a *composition* (with sum n) over S is a sequence of natural numbers from S that sum to n, as defined in Section 2.6. A *word* (with length k) over S is a sequence of k natural numbers from S. In later parts of this thesis, there will be much interaction between the sums and the lengths of such sequences, so we only use the term 'compositions', while specifying their attributes with adjectives *sum-n* or *length-k*.

The recurring theme in the study of compositions is enumeration. One seeks an explicit recurrence to count the number of compositions of a particular kind, and then aims to derive the generating function and the asymptotic expression of the recurrence.

There are various particular subsets of compositions to investigate. Examples include palindromic compositions, compositions with parts from a fixed set S, and Carlitz composi-

tions (where no adjacent parts are equal).

Pattern-avoidance is a major topic, where a *pattern* describes the relative sizes of parts in a substring. For example, the pattern 123 simply means a substring $\sigma_1\sigma_2\sigma_3$ such that $\sigma_1 < \sigma_2 < \sigma_3$, so in `2345`, both `234` and `345` are occurrences of the pattern 123. There have been many works on the compositions that contain or avoid fixed length-2 or length-3 patterns, as surveyed by Heubach and Mansour [61].

Nevertheless, the substrings studied in Sections 6.2 and 6.3 need to be exact matches. In the same example `2345`, the substring `123` does not occur at all. Therefore, the results on pattern-avoiding compositions cannot be applied to our context in Chapter 6.

3.3.1 The parking problem

One of the two motivations behind the original parking problem is a purely mathematical one, studied by Dvoretzky and Robbins [36] in 1964. On a stretch of an empty street-side without marked slots, cars of the same length sequentially arrive, and each parks at a position chosen uniformly at random from the available space. Eventually when there is no more gap to fit any more car, one wants to find the average number of cars that are parked along the street. This scenario thus led to the name 'parking problem'.

The other motivation behind this problem is physical chemistry, where the street-side corresponds to a surface, and the cars correspond to molecules that sequentially arrive and settle on this surface. This is the original setting studied by Jackson and Montroll [62], Rényi [99], Page [93] and Mackenzie [79] around 1960. A good survey on the history of these early works is given by Solomon and Weiner [109] in 1986.

Formulated in mathematical terms, the process starts with an interval $[0, n]$ corresponding to the street-side or the surface. The first unit-length segment—corresponding to a car or a molecule—uniformly chooses its left-endpoint to be a real number a_1 from the continuous interval $[0, n-1]$, and occupies the sub-interval $[a_1, a_1+1)$. Next, the second unit-length segment chooses a number a_2 uniformly from $[0, a_1-1) \cup [a_1+1, n]$, which represents all the

remaining possible left-endpoints; then this second segment occupies $[a_2, a_2 + 1)$.

This process repeats until there is no space to place any more unit-length segment. The next goal is to derive the function $M(n)$ that denotes the expected number of segments placed on the length-n interval at the end of the process. One also wants to derive the limit of $M(n)/n$ as $n \to \infty$. It turned out that this limit does not have a closed-form expression, and the numerical approximation to this limit is 0.74759. This number is known nowadays as Rényi's Parking Constant.

When molecules arrive on a surface with slots, they cannot simply settle anywhere, and have to fit with the slots. This motivates the discrete version of the same problem: instead of choosing from a continuous interval, a new molecule now chooses from a finite number of vacant slots.

In mathematical terms, the discrete version involves non-overlapping segments of length $a \in \mathbb{N}$ sequentially parking on the interval $[0, n]$. But instead of parking anywhere, each segment' left-endpoint must be an integer. Hence, the first segment uniformly chooses b_1 from the possible left-endpoints $\{0, 1, \ldots, n-a\}$, and occupies the sub-interval $[b_1, b_1 + a - 1]$. Next, the second segment uniformly choose b_2 from the remaining possible left-endpoints.

Similar to the continuous case, let $M(n)$ denote the expected number of segments, and investigate the limit $M(n)/n$ as $n \to \infty$. Unit-length segments lead to the trivial case, as the line is always completely filled. Nevertheless, when the $a = 2$, this limit turns out to have an elegant closed-form expression

$$\frac{1}{2} - \frac{1}{2e^2} \approx 0.43233.$$

When $a = 2$, the variance of the number of segments also has a closed-form expression. However, for larger segment-lengths, no closed-form expression seems possible, and one has to resort to numerical approximation as carried out by Mackenzie [79].

This discrete viewpoint with $a = 2$ is also relevant to graph theory in its pure sense. This

can be described as the *unfriendly seating problem*: people arrive at a long bench, uniformly choosing from the available spots that are not directly next to someone else. The background of unfriendly seating is summarised by Georgiou, Kranakis and Krizanc [46]. This article also generalises from the length-n line to the $m \times n$ grid. When $m = 1$, the $1 \times n$ grid is just the length-n line, with the results above. Then for $m = 2$, this article proves that when unfriendly people arrive to occupy the $2 \times n$ grid, the expected fraction of occupancy is

$$\frac{1}{2} - \frac{1}{4e} \approx 0.40803.$$

However, there is no closed-form result for larger values of m.

Note that when $a = 2$, the occupied seats form an independent set of the underlying graph. Using sophisticated techniques of random graphs, graph sequences and local limits, Krivelevich *et al.* [72] recently studied the same problem on a different underlying graphs such as trees.

Chapter 4

Centre and Median

The theme of this thesis is a graph-simplification method and its effects on the distance-function as well as the centre and the median. Before presenting the simplification method in Chapter 5, the basics of the centre and the median were defined in Section 2.4, and this present chapter summarises some results on the centres and medians in trees and k-trees. Some of these results come from existing works and are restated in uniform notation, while some are new.

Section 4.1 studies the centres in trees, beginning with unweighted trees: firstly, we state the well-known structural characterisation of the centre, and then discuss algorithms to locate the centre. Next, these results are generalised in Section 4.1.1 to edge-weighted trees, whose centres possess the same structural characterisation as those of unweighted trees, and we also present a similar centre-locating algorithm.

Section 4.2 studies the medians of trees, and its subsections focus respectively on unweighted trees, edge-weighted trees and vertex-weighted trees. Each of these three settings provides a structural characterisation of the medians and describes an algorithm to locate them.

4.1 Centres of Trees

The centres of unweighted trees has a very simple characterisation: the centre is either a single vertex or two adjacent vertices. This is a well-known fact, attested as early as in the 1869 article [63] by Camille Jordan. There are multiple arguments that can lead to this structural characterisation. Below we outline an argument based repeatedly removing the leaves of the tree. This leaf-removing procedure was first presented by Goldman [47, 48, 49], and later stated as Problem 6.21(a) in the book of exercises by Lovász [78]. In Section 4.1.1 we will outline another reasoning by Graham, Entringer and Székely [50].

Algorithm 4.1 Locating the centre of a tree

```
1: procedure LEAF-REMOVAL(a tree T)
2:     Let H = T                          ▷ H changes throughout the execution
3:     while H has more than two vertices do
4:         Identify all the leaves L(H) in the tree H
5:         Remove all the vertices in L(H) from H
6:     end while
7: return H
8: end procedure
```

Algorithm 4.1 is the leaf-removal procedure that locates the centre of a tree. Figure 4.1 shows an example run of Algorithm 4.1. At each iteration,all the leaves of the current tree are removed, until the remaining tree has no more than two vertices.

Let $L(T)$ denote the set of all leaves of a tree T, and consider the subgraph H induced by $T \setminus L(G)$. Then for any vertex $v \in H$,

$$\begin{aligned}
\mathrm{ecc}_T(v) &= \max\{d(v,x) \mid x \in T\} \\
&= \max\{d(v,x) \mid x \in L(T)\} \\
&= \max\{d(v,x) + 1 \mid x \in H\} \\
&= \max\{d(v,x) \mid x \in H\} + 1 \\
&= \mathrm{ecc}_H(v) + 1.
\end{aligned}$$

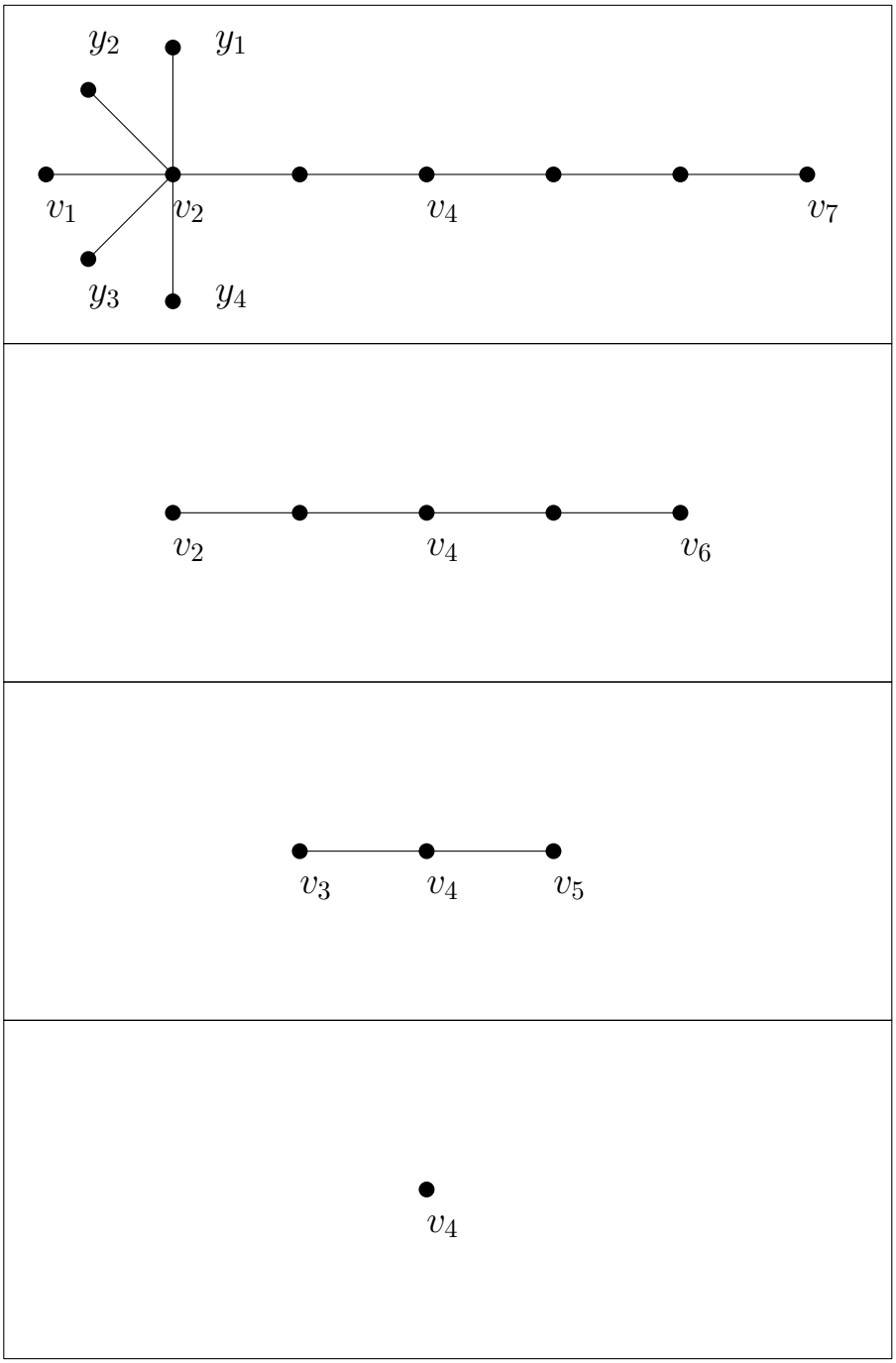


Figure 4.1: Locating the centre of an unweighted tree.

This means that v is in the centre of T if and only if v is in the centre of H. Therefore, after one iteration of leaf-removal, the centre of T stays exactly in H. This is an invariant which establishes the correctness of Algorithm 4.1.

The structural characterisation of a tree's centre follows from this algorithm. When the algorithm stops, the remaining tree consists of either one single vertex or two adjacent vertices. Each of these two trees is its own centre, and by the invariant above, it is the centre of the original tree.

Proposition 4.1. *In an unweighted tree, the centre consists of either a single vertex or two adjacent vertices.*

Note that the conceptually elegant Algorithm 4.1 runs in quadratic time, with the following reasoning. The number of iterations is $O(n)$; the worst case occurs when the tree has only two leaves at every iteration. Then within each iteration, identifying the leaves requires traversing all vertices once, and hence takes $O(n)$ time. Meanwhile, there can be at most $O(n)$ leaves at each iteration; assuming that deleting a vertex is a constant-time operation, it takes $O(n)$ time to delete all the leaves at each iteration. Therefore, the overall runtime is $O(n^2)$.

The quadratic-time Algorithm 4.1 was first presented by Goldman [49] in 1972. Soon afterwards, linear-time improvements in the unweighted case were presented by Handler [56] and Hedetniemi *et al.* [60]. Still, we only survey Algorithm 4.1 because it will be used in a later part of this thesis.

4.1.1 Centres of edge-weighted trees

This subsection generalises the centre-related results from unweighted trees to edge-weighted trees, where each edge is assigned a positive real number. To start with, we observe that the centre of an edge-weighted tree has the same structural characterisation as the centre of an unweighted tree. Nevertheless, as opposed to the leaf-removal reasoning used earlier, here in

the weighted case we adopt the pigeonhole-argument by Graham, Entringer and Székely [50, Theorem 2].

Proposition 4.2. *In an edge-weighted tree, the centre consists of either a single vertex or two adjacent vertices.*

Proof. For every vertex v, find an ecc-wit w, and mark the first edge of the path from v to w. Overall, n markings are carried out, but the tree only has $n-1$ edges, so by the pigeonhole principle, there must be an edge that is marked twice.

Let x and y be the endpoints of this twice-marked edge, and let S_x and S_y be the subtrees of x and y according to Definition 2.4. The next goal is to show that $\forall v \in T_x : \mathrm{ecc}(v) > \mathrm{ecc}(x)$ and $\forall v \in T_y : \mathrm{ecc}(v) > \mathrm{ecc}(y)$.

We marked the edge $\{x, y\}$ for x, which means that x must have an ecc-wit w in S_y. Consider any vertex $v \in S_x$ that is distinct from x; since v has to pass x to reach w, the equality $d(v, w) = d(v, x) + d(x, w)$ must hold. Next we show that w is also an ecc-wit of v.

For every vertex z in the whole tree, either of the following is true:

- $d(v, z) = d(v, x) + d(x, z)$ — v must pass x to reach z.

- $d(v, z) = d(x, z) - d(v, x)$ — the shortest path from v to z does not contain x.

Then for the first case,

$$\begin{aligned} d(v, w) &= d(v, x) + d(x, w) \\ &\geq d(v, x) + d(x, z) && (w \text{ is an ecc-wit of } x) \\ &= d(v, z) && (\text{1st case above}), \end{aligned}$$

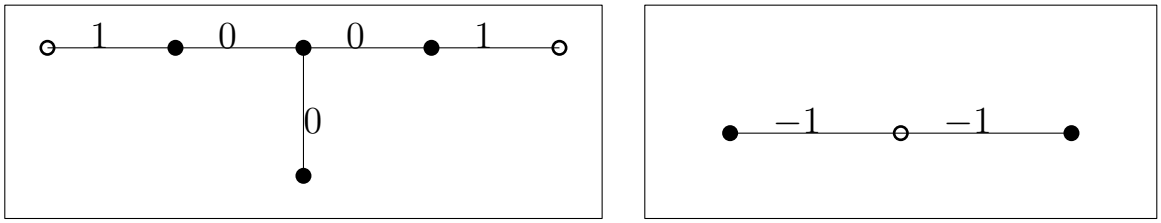


Figure 4.2: Centres of trees with non-positive edge-weights can have arbitrary shapes.

and for the second case,

$$\begin{aligned} d(v,w) &\geq d(v,x) + d(x,z) && \text{(same as the 1st case)} \\ &= d(v,x) + [d(v,z) + d(v,x)] && \text{(2st case above)} \\ &= 2 \cdot d(v,x) + d(v,z) \\ &> d(v,z) && \text{(weights are positive).} \end{aligned}$$

Therefore, as $d(v,w)$ is greater than or equal to $d(v,z)$ for any other vertex z, we have established that $d(v,w) = \text{ecc}(v)$ and that w is an ecc-wit of v. Since $d(v,w) > d(x,w)$, we conclude that $\text{ecc}(v) > \text{ecc}(x)$, and the same argument also yields $\text{ecc}(v) > \text{ecc}(y)$ for all $v \in S_y$.

Finally, $\text{ecc}(x)$ and $\text{ecc}(y)$ are strictly smaller than the eccentricity of all other vertices. If $\text{ecc}(x) = \text{ecc}(y)$, then the centre consists of two adjacent vertices. If one is strictly bigger than the other, then the centre consists of one vertex. □

Note that the theorem above no longer holds when the edge-weights are not limited to positive numbers. For example, the trees in Figure 4.2 have edges with 0 or negative weights, and their centres, indicated by the shaded vertices, can have arbitrary shapes and do not conform to the theorem above. This is another reason to restrict the edge-weights to the positive numbers; see the discussion on weighted graphs in Section 2.2.

Now we present an important lemma that will be crucial in a later chapter.

Theorem 4.3. *In a tree with positive edge-weights, the centre always lies on a diameter-path.*

Proof. Let $P = [v_1, \ldots, v_d]$ be a diameter-path, and consider a vertex x that is not on any diameter-path. (See Figure 4.3.) Let v_x be the vertex on P closest to x, and without loss of generality, assume that $d(v_1, v_x) \leq d(v_x, v_d)$. The goal is to show that $\text{ecc}(x) > \text{ecc}(v_x)$, which means that x can never belong to the centre.

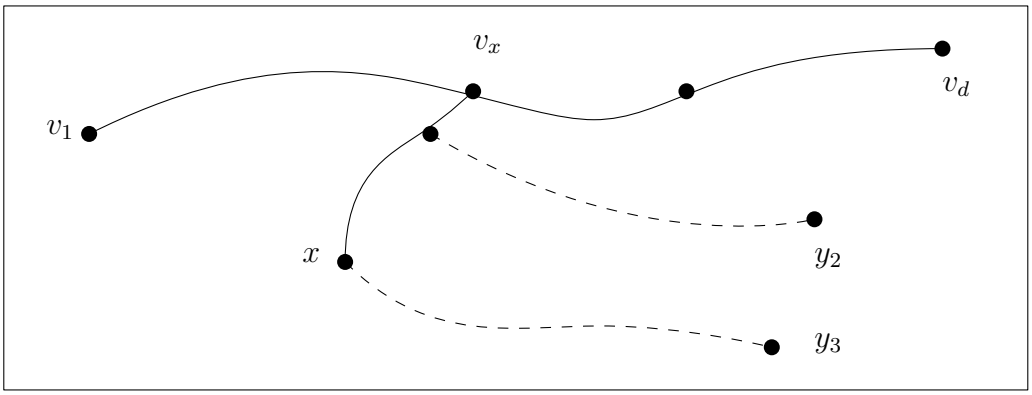


Figure 4.3: For the proof of Theorem 4.3.

It turns out that v_d must be an ecc-wit of x. If not, then y_1, y_2 or y_3 are possible ecc-wits of x, and in each case, we can construct a path that is longer than P, and contradict the assumption that P is a diameter-path. With a similar case-analysis, we can also show that v_d must also be an ecc-wit of v_x. Now, given $\text{ecc}(x) = d(x, v_d)$ and $\text{ecc}(v_x) = d(v_x, v_d)$, we can conclude that

$$\begin{aligned} \text{ecc}(v_x) &= d(v_x, v_d) \\ &< d(x, v_x) + d(v_x, v_d) \\ &= d(x, v_d) \\ &= \text{ecc}(x), \end{aligned}$$

which means that a vertex x not on any diameter-path necessarily has a larger eccentricity

and hence cannot be in the centre. Therefore, the centre in a weighted tree must be on a diameter-path. □

The rest of this section develops a modified leaf-removing algorithm to locate the centre of a weighted tree. First, we note that the unweighted Algorithm 4.1 does not work on weighted trees without modification. As a counter-example, Figure 4.4 shows a weighted tree, where the unlabelled edges have weight 1. If the weights are ignored, the centre is $\{z\}$. With the weighted considered, $\text{ecc}(x) = 6$, $\text{ecc}(y) = 5$ and $\text{ecc}(z) = 6$, so the centre is $\{y\}$. This shows that applying Algorithm 4.1 on this weighted tree does not produce the correct result. Hence, we entertain a few modifications.

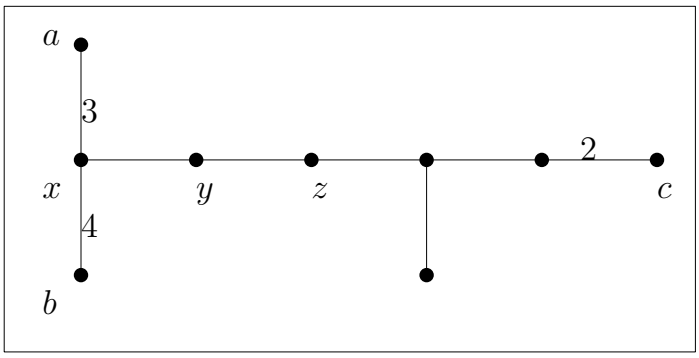


Figure 4.4: Weighted centre is $\{y\}$ but unweighted centre is $\{z\}$.

In an unweighted tree, all the eccentricities decrease by 1 when all the leaves are removed by Algorithm 4.1. However, this is not the case in a weighted tree because each leaf's associated edge can have different weights. In Figure 4.4, removing the leaf a reduces the distances by 3, removing b reduces them by 4, and removing c reduces them by 1. This suggests that we need a vertex-label function f to keep track of the distances to these removed vertices. Moreover, we need a modified version of the eccentricity function that takes f into account:

$$\text{ecc}'(v) = \max\{d(v, x) + f(x) \mid x \in T\}.$$

Note that initially when $f(v) = 0$ for all vertices v, the modified $\text{ecc}'(v)$ is the same as the original $\text{ecc}(v)$.

Algorithm 4.2 Locating the centre of an edge-weighted tree

```
1: procedure WEIGHTED-LEAF-REMOVAL(a weighted tree T)
2:     Let H = T
3:     Initialize f : T → ℕ_0 by setting f(v) = 0 for all v ∈ T
4:     repeat
5:         for each leaf v in H do
6:             Let p(v) be the neighbour of v          ▷ The leaf v has only one neighbour.
7:             Let h(v) = d(p(v), v) + f(v)            ▷ Potential new value of f(p(v)).
8:         end for
9:         for each leaf v with minimum h(v) do
10:            Remove v from H
11:            if h(v) > f(p(v)) then
12:                Update f(p(v)) = h(v)
13:            end if
14:        end for
15:    until H consists of one vertex, or two vertices with equal f value
16: return H
17: end procedure
```

Theorem 4.4. *Let H be the weighted tree at an intermediate step of Algorithm 4.2 on the original weighted tree T; that is, H is a weighted tree with at least three vertices and a vertex-labelling function f. Furthermore, let $v \in H$ be a leaf with the minimum value of $d(u, v) + f(v)$, where $u = p(v)$.*

Then $\mathrm{ecc}'(v) > \mathrm{ecc}'(u)$. In other words, v can never be in the centre and hence can be safely removed.

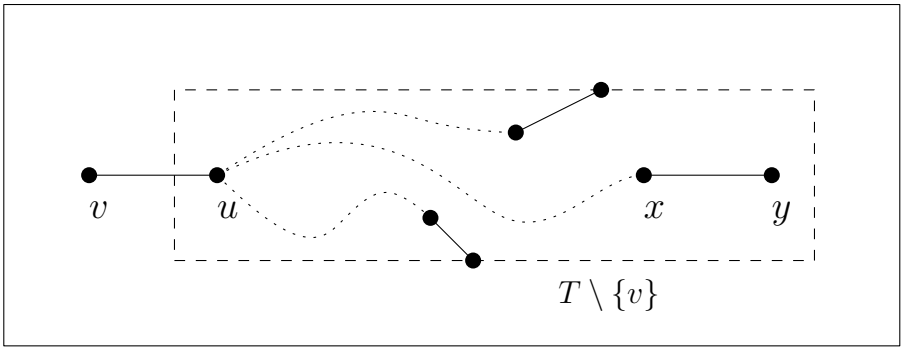


Figure 4.5: For the proof of Theorem 4.4.

Proof. Let y be another leaf in H with $x = p(y)$ such that

$$d(u, y) + f(y) = \max\{d(u, z) + f(z) \mid z \in H, z \neq v\},$$

which is illustrated by Figure 4.5. As v, u and y have to be distinct, this theorem requires the condition that H has at least three vertices. Then,

$$\begin{aligned} \mathrm{ecc}'(v) &= \max\{\ f(v)\ ,\ d(v, y) + f(y)\ \}, \text{ and} \\ \mathrm{ecc}'(u) &= \max\{\ h(v)\ ,\ d(u, y) + f(y)\ \}. \end{aligned}$$

Now, we look at $\mathrm{ecc}'(u)$ first:

$$\begin{aligned} d(u, y) + f(y) &= d(u, x) + d(x, y) + f(y) \\ &= d(u, x) + \big[d(x, y) + f(y)\big] \\ &= d(u, x) + h(y) \\ &\geq d(u, x) + h(v) && \text{(since } h(v) \text{ is minimum)} \\ &\geq h(v) && \text{(since } d(u, x) \text{ is positive)} \end{aligned}$$

This means that the second item in $\mathrm{ecc}'(u)$ is bigger than the first, so the maximum simply equals the second. That is, $\mathrm{ecc}'(u) = d(u, y) + f(y)$.

Next, $\mathrm{ecc}(v)' \geq d(v, y) + f(y)$ by the nature of the max function. But

$$\begin{aligned} d(v, y) + f(y) &= d(v, u) + d(u, y) + f(y) \\ &> d(u, y) + f(y) && (d(v, u) > 0) \\ &= \mathrm{ecc}'(u), \end{aligned}$$

so indeed $\mathrm{ecc}'(v) > \mathrm{ecc}'(u)$. □

We can now establish the correctness of Algorithm 4.2 as follows. By Theorem 4.4, the centre-vertices of the original T are never removed by the algorithm, so they are all contained in each intermediate weighted tree H. At the end of the algorithm, H is either a single vertex, or two adjacent vertices with equal f value, which must be the centre of the original T.

Similar to the unweighted Algorithm 4.1, the runtime of the weighted Algorithm 4.2 is also quadratic. The outer loop has $O(n)$ iterations, and the inner loops have $O(n)$ iterations each, so the overall runtime is $O(n^2)$.

The quadratic-time Algorithm 4.2 was first presented by Goldman [49] in 1972. The runtime was improved to $O(n \log n)$ by Kariv and Hakimi [65] in 1979, and then further improved to $O(n)$ by Megiddo [82] in 1983. The history of this line of research is also summarised in the relatively recent 2011 chapter by Tansel [114].

Example We now present an example of Algorithm 4.2 running on the weighted tree in Figure 4.6(i). To aid clarity, we omit the labels of the weight-one edges.

Initially, we assign the value $f(v) = 0$ to every vertex v. Then at the first step, we compare the values $h(v) = d(p(v), v) + f(v)$ of all the leaves $\{v_1, v_3, v_6, v_7\}$. Recall that for each leaf v, its only neighbour is denoted as $p(v)$ and is called the parent of v.

leaf	v_1	v_3	v_6	v_7
$h(v)$	3+0	4+0	3+0	1+0

Here v_7 has the smallest value, so we delete v_7 from the tree, resulting in (ii) in Figure 4.6. We also consider v_7's parent v_5; the value $f(v_5)$ is currently 0, which is smaller than $h(v_7) = 1$, so we update $f(v_5)$ to 1.

vertex	v_1	v_2	v_3	v_4	v_5	v_6
$f(v)$	0	0	0	0	**1**	0

[Step 2] We compare the remaining three leaves:

leaf	v_1	v_3	v_6
$h(v)$	3+0	4+0	3+0

Both v_1 and v_6 have the smallest value. Either can be deleted, and we arbitrarily choose v_6 to delete, resulting in Figure 4.6(iii). The parent of v_6 is v_5, and $f(v_5) = 1 < 3 = h(v_6)$, so we update $f(v_5)$ to 3.

vertex	v_1	v_2	v_3	v_4	v_5
$f(v)$	0	0	0	0	**3**

[Step 3] We compare the leaves in Figure 4.6(iii):

leaf	v_1	v_3	v_5
$h(v)$	3+0	4+0	1+3

Note that $h(v_5) = d(v_4, v_5) + f(v_5) = 1 + 3$. We remove v_1, and $f(v_2)$ is updated to 3.

vertex	v_2	v_3	v_4	v_5
$f(v)$	**3**	0	0	3

[Step 4] We look at Figure 4.6(iv), which contains only two leaves:

leaf	v_3	v_5
$h(v)$	4+0	1+3

Both have the smallest value, so we arbitrarily select v_3 to delete, and update $f(v_2)$ to 4.

vertex	v_2	v_4	v_5
$f(v)$	**4**	0	3

[Step 5] We compare:

leaf	v_2	v_5
$h(v)$	1+4	1+3

So we remove v_5 and update $f(v_4)$ to $h(v_5) = 4$.

vertex	v_2	v_4
$f(v)$	4	**4**

[Step 6] There are only two vertices left, so we exit the loop and return $\{v_2, v_4\}$ as the centre.

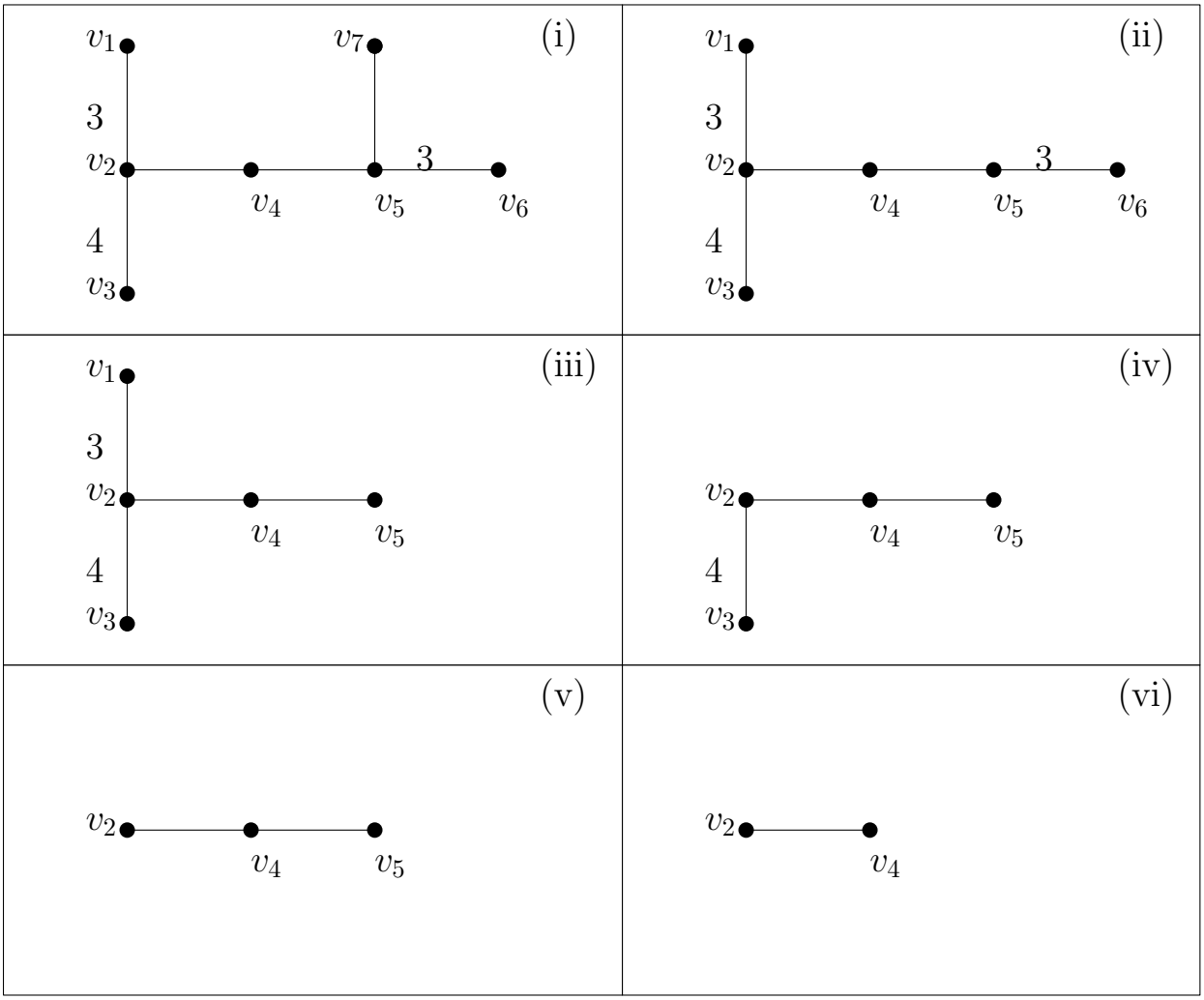


Figure 4.6: Locating the centre of an edge-weighted tree.

4.2 Medians of Trees

In this section, we turn our focus to the medians of unweighted trees, edge-weighted trees and vertex-weighted trees. In all these three settings, the medians consist of either a single vertex or two adjacent vertices, and can be located by similar algorithms. The distance-sum functions in the three settings have slightly different properties, which are summarised in Table 4.1 and will be discussed in more detail in the individual sections.

Table 4.1: Properties of the distance-sum in unweighted, edge-weighted and vertex-weighted trees, and their corresponding lemmas.

Unweighted trees	Edge-weighted trees	Vertex-weighted trees
$\mathrm{ds}(x) + \lvert S_x\rvert$ $= \mathrm{ds}(y) + \lvert S_y\rvert$	$\mathrm{ds}(x) + d(x,y) \cdot \lvert S_x\rvert$ $= \mathrm{ds}(y) + d(x,y) \cdot \lvert S_y\rvert$	$\mathrm{ds}^w(x) + f(S_x)$ $= \mathrm{ds}^w(y) + f(S_y)$
Lemma 4.5	Lemma 4.12	Lemma 4.17
Strictly convex	Not convex	Strictly convex
Lemma 4.8	Figure 4.9	Lemma 4.19

4.2.1 Medians of unweighted trees

We start with a fundamental lemma that relates the distance-sums of two adjacent vertices with the sizes of their subtrees.

Lemma 4.5. *Let x and y be two adjacent vertices in an unweighted tree, and let S_x and S_y be the subtrees of x and y according to Definition 2.4. Then*

$$\mathrm{ds}(x) + |S_x| = \mathrm{ds}(y) + |S_y|.$$

Equivalently,

$$\mathrm{ds}(x) = \mathrm{ds}(y) + n - 2|S_x|.$$

Proof. First, for all $u \in S_x$ and $v \in S_y$, we have $d(y,u) = d(x,u)+1$ and $d(x,v) = d(y,v)+1$ respectively. Then,

$$\begin{aligned}
\mathrm{ds}(x) + |S_x| &= |S_x| + \sum_{u \in S_x} d(x,u) + \sum_{v \in S_y} d(x,v) \\
&= \sum_{u \in S_x} \big[d(x,u)+1\big] + \sum_{v \in S_y} d(x,v) \\
&= \sum_{u \in S_x} \big[d(x,u)+1\big] + \sum_{v \in S_y} \big[d(y,v)+1\big] \\
&= \sum_{u \in S_x} d(y,u) + \sum_{v \in S_y} d(y,v) + |S_y| \\
&= \mathrm{ds}(y) + |S_y|.
\end{aligned}$$

The other equivalent expression is derived simply by using $|S_x| + |S_y| = n$ and then rearranging. □

This lemma then leads to the following corollaries.

Corollary 4.6. *If $x \sim y$, then $\mathrm{ds}(x) \leq \mathrm{ds}(y)$ if and only if $|S_x| \geq |S_y|$, or equivalently $|S_x| \geq n/2$.* □

Corollary 4.7. *If $x \sim y$ and $\mathrm{ds}(x) < \mathrm{ds}(y)$, then $\mathrm{ds}(x) < \mathrm{ds}(v)$ for all $v \in S_y$.*

Proof. Using Lemma 4.5, $\mathrm{ds}(x) < \mathrm{ds}(y)$ implies $|S_x| > n/2$. For every $v \in S_y$ that is adjacent to y, the subtree of y with respect to v contains S_x and thus has over $n/2$ vertices, so $\mathrm{ds}(y) < \mathrm{ds}(v)$. Since S_x already has more than $n/2$ vertices, the same argument extends to every vertex in S_y, and concludes that $\mathrm{ds}(x) < \mathrm{ds}(v)$ for every $v \in S_y$. □

The next lemma establishes that the distance-sum function in a tree is *strictly convex*, and this property will later help us characterise the structure of the medians of trees. This lemma also appears as Problem 6.22(a) in the book of exercises by Lovász [78].

Lemma 4.8. *Let x, y and z be vertices in a tree with $x \sim y \sim z$. Then* $2 \cdot \mathrm{ds}(y) < \mathrm{ds}(x) + \mathrm{ds}(z)$.

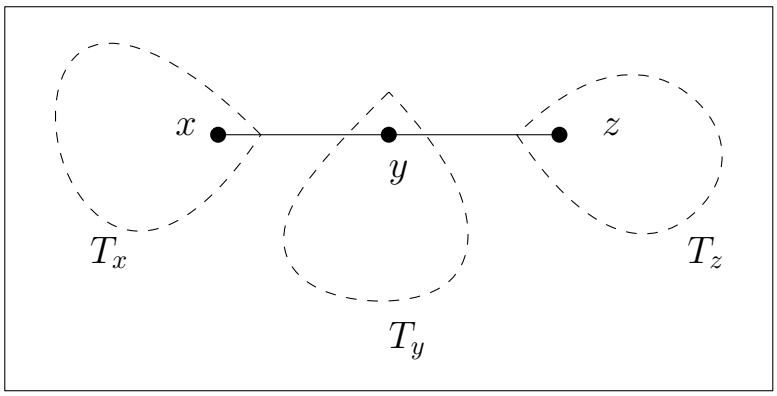


Figure 4.7: For the proof of distance-sum being strictly convex (Lemma 4.8).

Proof. Consider Figure 4.7, and let $|T_x| = k_x$ and $|T_z| = k_z$. Then Lemma 4.5 leads us to:

$$\mathrm{ds}(y) = \mathrm{ds}(x) + (2 \cdot k_x - n) \quad \text{and}$$
$$\mathrm{ds}(y) = \mathrm{ds}(z) + (2 \cdot k_z - n).$$

Adding these two equations together,

$$2 \cdot \mathrm{ds}(y) = \mathrm{ds}(x) + \mathrm{ds}(z) + 2(k_x + k_z - n).$$

However, after removing the subtrees T_x and T_z, we have at least one vertex y left-over, so $n - k_x - k_z \geq 1$. Consequently, $k_x + k_z - n \leq -1$. Therefore,

$$\begin{aligned} 2 \cdot \mathrm{ds}(y) &\leq \mathrm{ds}(x) + \mathrm{ds}(z) + 2 \cdot (-1) \\ &< \mathrm{ds}(x) + \mathrm{ds}(z), \end{aligned}$$

which completes the proof. □

Now we use the strict convexity of the distance-sum to establish the structural characterisation of the medians of unweighted trees.

Theorem 4.9. *In an unweighted tree, the median consists of either a single vertex or two adjacent vertices.*

Proof. Firstly, observe that both one-vertex and two-vertex medians are possible. Then we proceed to prove that the median cannot be disconnected or have more than two vertices. Assume that x and y are two non-adjacent median-vertices, and let

$$x = v_0 \sim v_1 \sim \cdots v_p = y.$$

Now $v_0 = x$ has the minimum value, so $\mathrm{ds}(v_1) \geq \mathrm{ds}(v_0)$, and hence:

$$\begin{aligned} \mathrm{ds}(v_0) + \mathrm{ds}(v_2) &> 2 \cdot \mathrm{ds}(v_1) && \text{(by Lemma 4.8)} \\ &\geq \mathrm{ds}(v_0) + \mathrm{ds}(v_1). \end{aligned}$$

Cancelling $\mathrm{ds}(v_0)$ leads to $\mathrm{ds}(v_2) > \mathrm{ds}(v_1) \geq \mathrm{ds}(v_0)$, so

$$\begin{aligned} \mathrm{ds}(v_1) + \mathrm{ds}(v_3) &> 2 \cdot \mathrm{ds}(v_2) && \text{(by Lemma 4.8)} \\ &> \mathrm{ds}(v_0) + \mathrm{ds}(v_1) \end{aligned}$$

which leads to $\mathrm{ds}(v_3) > \mathrm{ds}(v_0)$. Repeating the same argument, we eventually obtain $\mathrm{ds}(v_p) > \mathrm{ds}(v_0)$, which is the same as $\mathrm{ds}(y) > \mathrm{ds}(x)$. However, this contradicts with the assumption of $\mathrm{ds}(x)$ and $\mathrm{ds}(y)$ being both minimum and hence equal. Therefore, no two median-vertices x and y can be distance two apart, which completes the proof. □

The rest of this section describes an algorithm to compute the distance-sums of all the vertices in a tree. After computing all of the distance-sums, we can then easily locate the median. The backbone is Lemma 4.5, which allows us to propagate a known value $\mathrm{ds}(y)$

across an edge, and obtain the distance-sum of the neighbour x, provided that we also know the subtree-size $|S_x|$. The subtree-size is easy to compute in a rooted tree, so in what follows, we will root the input tree at an arbitrary vertex. The choice of the root ends up having no effect to the the overall result.

We first prepare some fundamental notions related to rooted trees, and recall the definitions of the *subtree* and the *children* of a vertex from Section 2.5.

Definition 4.10. Let T be a rooted tree, with root r. The *restricted distance-sum* of a vertex x, denoted $\mathrm{ds}_r(x)$, is the sum of the distances from x to all the vertices in the subtree of x. In other terms,

$$\mathrm{ds}_r(x) = \sum_{v \in S_x} d(x, v).$$

The subtree of the root is the whole tree itself, so the root's overall distance-sum $\mathrm{ds}(r)$ simply equals to its restricted distance-sum $\mathrm{ds}_r(r)$.

The subtree-size of x can be calculated from the subtree-sizes of the children of x. That is, $|S_x| = 1 + \sum_{y \in \mathrm{Ch}(x)} |S_y|$. In other words, we can recursively compute the subtree-sizes from the leaves to the root. The restricted distance-sums can also be calculated in a similar way, as the next proposition explains.

Proposition 4.11. *For any vertex x in the tree,*

$$\mathrm{ds}_r(x) = \sum_{y \in \mathrm{Ch}(x)} \big[|S_y| + ds_r(y) \big].$$

Proof. This equation can be derived as follows.

$$\begin{aligned} ds_r(x) &= \sum_{v \in S_x} d(x,v) \\ &= \sum_{y \in \mathrm{Ch}(x)} \sum_{v \in S_y} d(x,v) \\ &= \sum_{y \in \mathrm{Ch}(x)} \sum_{v \in S_y} \big[d(x,y) + d(y,v) \big] \\ &= \sum_{y \in \mathrm{Ch}(x)} \sum_{v \in S_y} \big[1 + d(y,v) \big] \\ &= \sum_{y \in \mathrm{Ch}(x)} \big[|S_y| + \sum_{v \in S_y} d(y,v) \big] \\ &= \sum_{y \in \mathrm{Ch}(x)} \big[|S_y| + \mathrm{ds}_r(y) \big]. \end{aligned}$$

□

Now, Algorithm 4.3 is the method to compute all the distance-sums of a tree. After choosing an arbitrary vertex as the root, this algorithm then consists of two stages.

The first stage, called the *up-propagation*, works level by level from bottom to top, and calculates the subtree-size and the restricted distance-sum of each vertex on each level using Proposition 4.11. When the vertex v is a leaf, Lines 5 and 6 become empty sums because a leaf has no child, so $|S_v| = 1$ and $\mathrm{ds}_r(v) = 0$.

After the first stage, every vertex x holds both $\mathrm{ds}_r(x)$ and $|S_x|$. The second stage, called the *down-propagation*, works from the root to the leaves. For every child v of the root, $\mathrm{ds}(v)$ is calculated using Lemma 4.5, given $\mathrm{ds}(r)$ and $|S_v|$ computed by the first stage. Repeating this process level by level from top to bottom, we obtain the distance-sum of every vertex.

The up-propagation stage traverses every vertex once, and so does the down-propagation stage. Therefore, Algorithm 4.3 runs in linear time.

Algorithm 4.3 Computing the distance-sums in a tree

procedure TreeDS(a tree T with the root r)
  $h \leftarrow$ the deepest level of T
  for each i ranging from h to 0 **do** ▷ the up-propagation
    for each vertex v on the ith level **do**
      $|S_v| \leftarrow 1 + \sum_{x \in \mathrm{Ch}(v)} |S_x|$
      $\mathrm{ds}_r(v) \leftarrow \sum_{x \in \mathrm{Ch}(v)} \big(\mathrm{ds}_r(x) + |S_x|\big)$ ▷ by Proposition 4.11
    end for
  end for
  Now we have $\mathrm{ds}(r)$ and $|S_v|$ for every vertex v
  for each i ranging from 1 to h **do** ▷ the down-propagation
    for each vertex v on the ith level **do**
      $u \leftarrow$ the parent of v
      $\mathrm{ds}(v) \leftarrow \mathrm{ds}(u) + n - 2 \cdot |S_v|$ ▷ by Lemma 4.5
    end for
  end for
end procedure

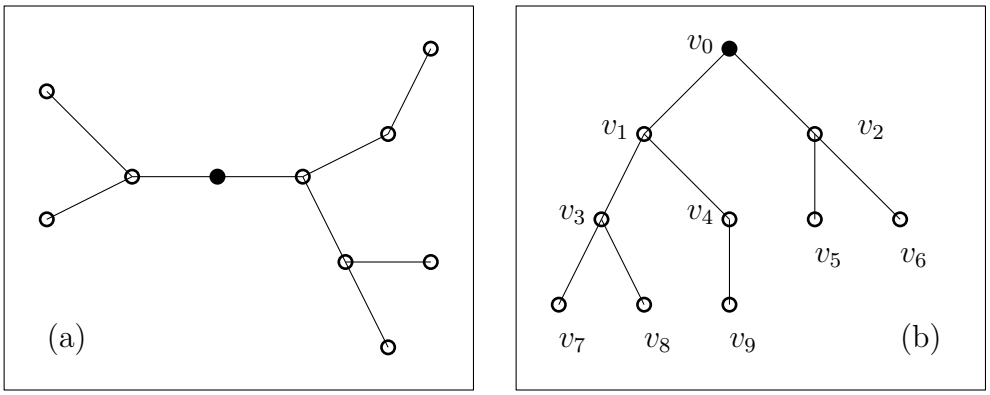


Figure 4.8: Computing the distance-sums in a tree.

Example. To compute the distance-sum of every vertex of any unrooted tree (like the one in Figure 4.8a), we first choose an arbitrary vertex (in this case the shaded one) and make this the root. Now that we have a rooted tree (Figure 4.8b), we can apply Algorithm 4.3. This algorithm begins with the stage of up-propagation. Firstly, every leaf is initialized with subtree-size 1 and subtree-ds 0.

vertex	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$\lvert S_v \rvert$						1	1	1	1	1
$\mathrm{ds}_r(v)$						0	0	0	0	0

The the deepest level of the rooted tree is already filled, so we deal with the second level[1] next. For v_3 whose children are v_7 and v_8,

$$|S_{v_3}| = 1 + |S_{v_7}| + |S_{v_8}| = 1 + 1 + 1 = 3, \text{ and}$$

$$\mathrm{ds}_r(v_3) = \big(|S_{v_7}| + \mathrm{ds}_r(v_7)\big) + \big(|S_{v_8}| + \mathrm{ds}_r(v_8)\big) = (1 + 0) + (1 + 0) = 2.$$

Meanwhile, v_4 has v_9 as its only child, so $|S_{v_4}|$ and $\mathrm{ds}_r(v_4)$ are 2 and 1 respectively.

vertex	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$\lvert S_v \rvert$				**3**	**2**	1	1	1	1	1
$\mathrm{ds}_r(v)$				**2**	**1**	0	0	0	0	0

Next are the vertices v_1 and v_2 on the first level.

$$|S_{v_1}| = 1 + |S_{v_3}| + |S_{v_4}| = 1 + 3 + 2 = 6,$$

$$\mathrm{ds}_r(v_1) = \big(|S_{v_3}| + \mathrm{ds}_r(v_3)\big) + \big(|S_{v_4}| + \mathrm{ds}_r(v_4)\big) = (3 + 2) + (2 + 1) = 8,$$

and the values of v_2 are computed identically to those of v_3.

[1] The root is on level zero.

vertex	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$\lvert S_v\rvert$		**6**	**3**	3	2	1	1	1	1	1
$\mathrm{ds}_r(v)$		**8**	**2**	2	1	0	0	0	0	0

Finally we compute the root's values. Note that $|S_{v_0}|$ equals n, the total number of vertices, and also $\mathrm{ds}_r(v_0) = \mathrm{ds}(v_0)$.

$$|S_{v_0}| = 1 + |S_{v_1}| + |S_{v_2}| = 1 + 6 + 3 = 10,$$

$$\mathrm{ds}_r(v_0) = \big(|S_{v_1}| + \mathrm{ds}_r(v_1)\big) + \big(|S_{v_2}| + \mathrm{ds}_r(v_2)\big) = (6 + 8) + (3 + 2) = 19.$$

vertex	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$\lvert S_v\rvert$	**10**	6	3	3	2	1	1	1	1	1
$\mathrm{ds}_r(v)$	**19**	8	2	2	1	0	0	0	0	0

Now begins the second stage (down-propagation). Given the values of the root, we start with the first-level vertices v_1 and v_2.

$$\mathrm{ds}(v_1) = \mathrm{ds}(v_0) + n - 2 \times |S_{v_1}| = 19 + 10 - 2 \times 6 = 17, \text{ and}$$
$$\mathrm{ds}(v_2) = \mathrm{ds}(v_0) + n - 2 \times |S_{v_2}| = 19 + 10 - 2 \times 3 = 23.$$

vertex	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$\mathrm{ds}(v)$	19	**17**	**23**							

Next are the vertices v_3 to v_6 on the second level.

$$\mathrm{ds}(v_3) = \mathrm{ds}(v_1) + n - 2 \times |S_{v_3}| = 17 + 10 - 2 \times 3 = 21,$$
$$\mathrm{ds}(v_4) = \mathrm{ds}(v_1) + n - 2 \times |S_{v_4}| = 17 + 10 - 2 \times 2 = 23,$$
$$\mathrm{ds}(v_5) = \mathrm{ds}(v_2) + n - 2 \times |S_{v_5}| = 23 + 10 - 2 \times 1 = 31,$$
$$\mathrm{ds}(v_6) = \mathrm{ds}(v_2) + n - 2 \times |S_{v_6}| = 23 + 10 - 2 \times 1 = 31.$$

The steps for the final three vertices will be omitted, and the following table shows the final result.

vertex	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$\mathrm{ds}(v)$	19	17	23	21	23	31	31	29	29	31

After computing all the distance-sums, we can then search through the vertices one last time to determine that the median of the tree in Figure 4.8 is $\{v_1\}$.

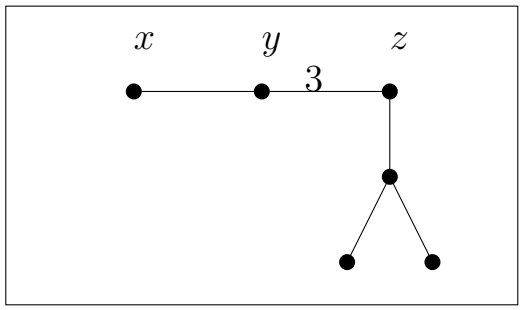


Figure 4.9: Distance-sum not convex in edge-weighted trees.

4.2.2 Medians of edge-weighted trees

In edge-weighted trees, the distance-sum is not convex, and hence not strictly convex. Figure 4.9 shows a counter-example, where the edge $\{y, z\}$ has weight 3, and all remaining edges have weight 1. In this tree, $\mathrm{ds}(y) = 14$ and $\mathrm{ds}(x) + \mathrm{ds}(z) = 37$, so $2 \cdot \mathrm{ds}(y) > \mathrm{ds}(x) + \mathrm{ds}(z)$, which violates the criterion of convexity. Nevertheless, we can still characterise the structure of the median using similar lemmas and a different reasoning.

First, the following lemma is the analogous version of the unweighted Lemma 4.5. If we replace each occurrence of $d(x, y)$ by 1, then we end up with Lemma 4.5 exactly. Indeed, any unweighted graph can be viewed as an edge-weighted graph in which every edge has weight one.

Lemma 4.12. *For any two adjacent vertices x and y in an edge-weighted tree,*

$$\mathrm{ds}(x) + d(x, y) \cdot |S_x| = \mathrm{ds}(y) + d(x, y) \cdot |S_y|.$$

Equivalently, $\mathrm{ds}(x) = \mathrm{ds}(y) + d(x, y) \cdot (n - 2|S_x|)$.

Proof. Firstly $\forall u \in S_x : d(y, u) = d(u, x) + d(x, y)$ and $\forall v \in S_y : d(x, v) = d(x, y) + d(y, v)$. Then,

$$\begin{aligned}
\mathrm{ds}(x) &= \sum_{u \in S_x} d(u, x) + \sum_{v \in S_y} d(x, v) \\
&= \sum_{u \in S_x} d(u, x) + \sum_{v \in S_y} \Big(d(x, y) + d(y, v) \Big) \\
&= \sum_{u \in S_x} d(u, x) + \sum_{v \in S_y} d(x, y) + \sum_{v \in S_y} d(y, v) \\
&= \sum_{u \in S_x} d(u, x) + \Big(d(x, y) \cdot |S_y| \Big) + \sum_{v \in S_y} d(y, v).
\end{aligned}$$

Similarly,

$$\mathrm{ds}(y) = \sum_{v \in S_y} d(y, v) + \Big(d(x, y) \cdot |S_x| \Big) + \sum_{u \in S_x} d(u, x).$$

Subtracting these two equations and rearranging leads to the goal of

$$\mathrm{ds}(x) + d(x,y) \cdot |S_x| = \mathrm{ds}(y) + d(x,y) \cdot |S_y|.$$

Finally, the equivalent expression is derived simply by replacing $|S_y|$ with $n - 2|S_x|$. □

This lemma has an elegant implication. Given two adjacent vertices x and y, to determine which of $\mathrm{ds}(x)$ and $\mathrm{ds}(y)$ is larger, we can simply compare $|S_x|$ and $|S_y|$ without taking the edge-weight $d(x,y)$ into account.

Corollary 4.13. *For every adjacent pair of vertices x and y in a weighted tree,* $\mathrm{ds}(x) \leq \mathrm{ds}(y)$ *if and only if* $|S_x| \geq |S_y|$.

Proof. Firstly, we rearrange the equation of Lemma 4.12:

$$\mathrm{ds}(x) - \mathrm{ds}(y) = d(x,y) \cdot \big(|S_y| - |S_x|\big).$$

Then the inequality $\mathrm{ds}(x) \leq \mathrm{ds}(y)$ is equivalent to $\mathrm{ds}(x) - \mathrm{ds}(y) \leq 0$, and applying this to the rearranged equation above:

$$d(x,y) \cdot \big(|S_y| - |S_x|\big) \leq 0.$$

Since the weights are always positive, we can divide both sides by $d(x,y)$ and obtain $|S_y| - |S_x| \leq 0$, which is equivalent to $|S_x| \geq |S_y|$. □

Theorem 4.14. *In an edge-weighted tree, the median is either a single vertex or two adjacent vertices.*

Proof. Assume there are two median-vertices v_1 and v_p separated by at least one vertex. For each $j \in [p]$, let k_j be the number of vertices in the subtree of v_j, as illustrated in the figure below. In this setting, every k_j is at least 1.

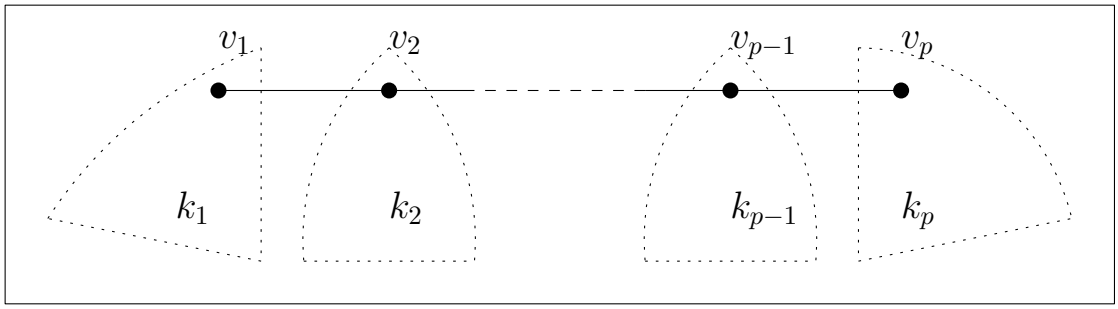


Figure 4.10: Assuming v_1 and v_p are median-vertices separated by at least one vertex.

The vertex v_1 is assumed to be have the minimum distance-sum, so $\mathrm{ds}(v_1) \leq \mathrm{ds}(v_2)$, and by Corollary 4.13, $k_1 \geq k_2 + \cdots + k_{p-1} + k_p$. Similarly for v_k, we have $k_p \geq k_1 + k_2 + \cdots + k_{p-1}$. Next, we add these two inequalities together:

$$k_1 + k_p \geq k_1 + k_p + 2 \cdot (k_2 + \cdots + k_{p-1})$$
$$0 \geq k_2 + \cdots + k_{p-1}.$$

However, this is impossible because every k_j is at least 1. Therefore, the assumption of such v_1 and v_p is contradicted, so the median is either a single vertex or two adjacent vertices. □

Now we turn our attention from the structural to the algorithmic. On an edge-weighted tree, we can compute the distance-sum of every vertex with an algorithm similar to the unweighted Algorithm 4.3, using the following proposition.

Proposition 4.15. *For any vertex x in a weighted rooted tree,*

$$\mathrm{ds}_r(x) = \sum_{y \in \mathrm{Ch}(x)} \big[d(x,y) \cdot |S_y| + \mathrm{ds}_r(y) \big].$$

Algorithm 4.4 Computing the distance-sums in an edge-weighted tree

```
1: procedure WeightedTreeDS(a tree T with the root r)
2:     h ← the deepest level of T
3:     for each i ranging from h to 0 do                    ▷ the up-propagation
4:         for each vertex v on the ith level do
5:             |S_v| ← 1 + Σ_{x∈Ch(v)} |S_x|
6:             ds_r(v) ← Σ_{x∈Ch(v)} (ds_r(x) + d(v,x) · |S_x|)    ▷ by Proposition 4.15
7:         end for
8:     end for
9:     Now we have ds(r) and |S_v| for every vertex v
10:    for each i ranging from 1 to h do                    ▷ the down-propagation
11:        for each vertex v on the ith level do
12:            u ← the parent of v
13:            ds(v) ← ds(u) + d(u,v) · (n − 2 · |S_v|)     ▷ by Lemma 4.12
14:        end for
15:    end for
16: end procedure
```

Proof. Similar to the earlier counterpart of Proposition 4.11,

$$\begin{aligned}
\mathrm{ds}_r(x) &= \sum_{v \in S_x} d(x,v) \\
&= \sum_{y \in \mathrm{Ch}(x)} \sum_{v \in S_y} d(x,v) \\
&= \sum_{y \in \mathrm{Ch}(x)} \sum_{v \in S_y} \big[d(x,y) + d(y,v) \big] \\
&= \sum_{y \in \mathrm{Ch}(x)} \sum_{v \in S_y} \big[d(x,y) + d(y,v) \big] \\
&= \sum_{y \in \mathrm{Ch}(x)} \big[d(x,y) \cdot |S_y| + \sum_{v \in S_y} d(y,v) \big] \\
&= \sum_{y \in \mathrm{Ch}(x)} \big[d(x,y) \cdot |S_y| + \mathrm{ds}_r(y) \big].
\end{aligned}$$

□

Lemma 4.12 and Proposition 4.15 justify Algorithm 4.4, which computes all the distance-sums in a weighted tree. The only differences with the unweighted version Algorithm 4.3 lie on Lines 6 and 13.

4.2.3 Medians of vertex-weighted trees

Later in Section 5.5, we will deal with vertex-weighted trees and their medians, so we lay down some required foundations in this present section.

Definition 4.16. A *vertex-weighted graph* G is a graph with a *vertex-weight* function $f : V(G) \to \mathbb{R}^+$. The *vertex-weighted distance-sum* of each vertex x is defined to be

$$\mathrm{ds}^w(x) = \sum_{v \in G} d(x, v) \cdot f(v).$$

Then the *median* of a vertex-weighted graph is the set of vertices that minimise the vertex-weighted distance-sum. In addition, the *weight of a vertex-subset* S is defined to be the sum of the vertices' weights:

$$f(S) = \sum_{v \in S} f(v).$$

Note that if G is a vertex-weighted graph, then the term 'median' of G always refers to the median with respect to the vertex-weighted distance-sum. Alternatively, if a graph does not have vertex-weights, then we can view it as a vertex-weighted graph with every vertex having weight 1, and $f(S)$ is the same as the cardinality of S. Next, we generalise Lemma 4.5 to accommodate vertex-weights (noting the difference with the edge-weighted version in Section 4.2.2).

Lemma 4.17. *Let x and y be adjacent vertices in a vertex-weighted tree, and define S_x and S_y according to Definition 2.4. Then*

$$\mathrm{ds}^w(x) + f(S_x) = \mathrm{ds}^w(y) + f(S_y).$$

Proof. The reasoning is almost identical to the proof of Lemma 4.5. We begin by deriving

$\mathrm{ds}^w(x)$:

$$\begin{aligned}
\mathrm{ds}^w(x) &= \sum_{v\in G} d(x,v)\cdot f(v) \\
&= \sum_{v\in S_x} \big[d(x,v)\cdot f(v)\big] + \sum_{v\in S_y} \big[d(x,v)\cdot f(v)\big] \\
&= \sum_{v\in S_x} \big[d(x,v)\cdot f(v)\big] + \sum_{v\in S_y} \big[[d(x,y)+d(y,v)]\cdot f(v)\big] \\
&= \sum_{v\in S_x} \big[d(x,v)\cdot f(v)\big] + \sum_{v\in S_y} \big[[1+d(y,v)]\cdot f(v)\big] \\
&= \sum_{v\in S_x} \big[d(x,v)\cdot f(v)\big] + \sum_{v\in S_y} \big[f(v)+d(y,v)\cdot f(v)\big] \\
&= \sum_{v\in S_x} \big[d(x,v)\cdot f(v)\big] + \sum_{v\in S_y} f(v) + \sum_{v\in S_y} [d(y,v)\cdot f(v)] \\
&= \sum_{v\in S_x} \big[d(x,v)\cdot f(v)\big] + f(S_y) + \sum_{v\in S_y} \big[d(y,v)\cdot f(v)\big].
\end{aligned}$$

The exact same argument also yields:

$$\mathrm{ds}^w(y) = \sum_{v\in S_x} [d(x,v)\cdot f(v)] + f(S_x) + \sum_{v\in S_y} [d(y,v)\cdot f(v)].$$

Therefore, after subtracting these two equations and rearranging, we obtain the lemma's statement. □

The lemma above implies that the median of the vertex-weighted tree can be computed in linear time using a modified version of Algorithm 4.3, where every vertex's distance-sum $\mathrm{ds}(v)$ is replaced by $\mathrm{ds}^w(v)$, and $|S_v|$ is replaced by the weight $f(S_v)$.

Corollary 4.18. *With x, y, S_x, S_y defined in the same way, Lemma 4.17 above directly implies the following statements.*

(i) $\mathrm{ds}^w(x) = \mathrm{ds}^w(y) + 2\cdot f(S_y) - f(T)$.

(ii) If $\mathrm{ds}^w(x) \le \mathrm{ds}^w(y)$, *then* $f(S_x) \ge f(S_y)$. □

In the previous sections, we have shown that the distance-sum is strictly convex in unweighted trees, but is not convex in edge-weighted trees. In vertex-weighted trees, the distance-sum turns out to be strictly convex, as shown by the following lemma.

Lemma 4.19. *The vertex-weighted distance-sum is strictly convex. In other terms, if* $x \sim y \sim z$, *then* $2 \cdot \mathrm{ds}^w(y) < \mathrm{ds}^w(x) + \mathrm{ds}^w(z)$.

Proof. The reasoning is almost exactly identical with the proof of Lemma 4.8. Consider Figure 4.7, and let $f(T_x) = k_x$ and $f(T_z) = k_z$. Then by Corollary 4.18(i),

$$\begin{aligned} \mathrm{ds}^w(y) &= \mathrm{ds}^w(x) + (2 \cdot k_x - f(T)) \quad \text{and} \\ \mathrm{ds}^w(y) &= \mathrm{ds}^w(z) + (2 \cdot k_z - f(T)). \end{aligned}$$

Adding these two equations together,

$$2 \cdot \mathrm{ds}^w(y) = \mathrm{ds}^w(x) + \mathrm{ds}^w(z) + 2(k_x + k_z - f(T)).$$

However, $f(T) - k_x - k_z > 0$ because the vertex-weights are all positive, so

$$\begin{aligned} 2 \cdot \mathrm{ds}^w(y) &< \mathrm{ds}^w(x) + \mathrm{ds}^w(z) + 2 \cdot 0 \\ &< \mathrm{ds}^w(x) + \mathrm{ds}^w(z), \end{aligned}$$

which completes the proof. □

Now that the distance-sum in vertex-weighted trees are strictly convex, we can use the same argument in Theorem 4.9 to prove the structural characterisation of the medians of vertex-weighted trees.

Corollary 4.20. *The median of a vertex-weighted tree consists of either one vertex or two adjacent vertices.* □

We now present some additional facts that will assist us when we deal with vertex-weighted trees later in Section 5.5.

Lemma 4.21. *Let x, y, S_x, S_y be defined in the same way as in Lemma 4.17. If $\mathrm{ds}^w(x) \leq \mathrm{ds}^w(y)$, then for all $v \in S_y : \mathrm{ds}^w(x) < \mathrm{ds}^w(v)$.*

Proof. Consider any $v \in S_y$ such that $y \sim v$. Now,

$$\begin{aligned} 2 \cdot \mathrm{ds}^w(y) &< \mathrm{ds}^w(x) + \mathrm{ds}^w(v) && \text{(strict convexity)} \\ &\leq \mathrm{ds}^w(y) + \mathrm{ds}^w(v) && \text{(this lemma's premise),} \end{aligned}$$

which means that $\mathrm{ds}^w(y) < \mathrm{ds}^w(v)$. Then, repeating the same reasoning on y and z allows us to propagate the inequality into the whole of S_y. □

Corollary 4.22. *If $\mathrm{ds}^w(x) = \mathrm{ds}^w(y)$, then $\{x, y\}$ must be the median.* □

4.2.4 Convexity

We conclude the medians of trees with a discussion on convexity and strict convexity. A numeric function f on the vertices of a graph is called *convex* when $2 \cdot f(y) \leq f(x) + f(z)$ for all vertices x, y, z with $x \sim y$ and $y \sim z$. In contrast, f is called *strictly convex* when the inequality is strict: $2 \cdot f(y) < f(x) + f(z)$.

We showed that the $\mathrm{ds}(x)$ function in unweighted trees and the $\mathrm{ds}^w(x)$ function in vertex-weighted trees are strictly convex. Then using the strict convexity, we further established that the medians in unweighted trees and vertex-weighted trees consist of either a single vertex or two adjacent vertices. For edge-weighted trees, although the distance-sum is not convex, we still arrived at the same structural characterisation via a different route.

Now, one might wonder about the convexity of the eccentricity function in unweighted and edge-weighted trees. It turns out that the eccentricity in an unweighted tree is convex but not strictly convex, and this applies to edge-weighted trees too. As a result, the structure

of the centres in both unweighted and edge-weighted trees cannot be characterised using the argument in Theorem 4.9, so we have to resort to other methods.

To show the convexity of the eccentricity function in an unweighted tree, one can use the identical reasoning as in Lovász [78, 6.21b]. On the other hand, Figure 4.11 shows an unweighted tree that is not strictly convex; the numbers in the figure are the eccentricities, and $2 \cdot \mathrm{ecc}(v_4) = \mathrm{ecc}(v_3) + \mathrm{ecc}(v_5)$, which does not satisfy the criterion of strict convexity.

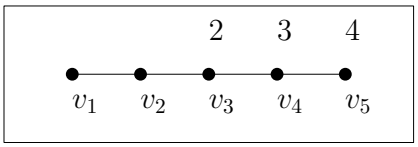


Figure 4.11: Eccentricity not strictly convex in a tree.

In conclusion:

- The eccentricity in an unweighted or edge-weighted tree is convex [78, 6.21b] but not strictly convex (Figure 4.11).
- The distance-sum in an unweighted tree is strictly convex (Lemma 4.8).
- The distance-sum in an edge-weighted tree is not convex, and hence not strictly convex (Figure 4.9).
- The distance-sum in a vertex-weighted tree is strictly convex (Lemma 4.19).

4.3 Conclusion and Outlook

This chapter summarised the structural and algorithmic results on the centres and the medians of unweighted and weighted trees.

In Section 4.1 we began with the leaf-removing Algorithm 4.1 that locates the centre of an unweighted tree. This algorithm gives us a way to establish the well-known structural characterisation that the centre of a tree is either a single vertex or two adjacent vertices.

In practice, Algorithm 4.1 runs in quadratic time, but there are linear-time algorithms by Handler [56] and Hedetniemi *et al.* [60] for locating the centre of a tree.

Next, in Section 4.1.1 we extended the results on the centre from unweighted to edge-weighted trees. Firstly, Proposition 4.2 followed a different reasoning to establish the same structural characterisation that the centre of a weighted tree also is either a single vertex or two adjacent vertices. It is necessary for the edge-weights to be positive, as Figure 4.2 showed counter-examples of trees with non-positive edge-weights, whose centres do not conform to the structural characterisation. We also developed a weighted analogue of Algorithm 4.1, and presented Algorithm 4.2 that locates the centre of a weighted tree by repeatedly removing vertices. Algorithm 4.2 runs in quadratic time, but to locate the centre of a weighted tree, there is a linear-time algorithm by Megiddo [82].

Section 4.2 look at the medians of unweighted trees, edge-weighted trees and vertex-weighted trees. In the unweighted case, Theorem 4.9 restated the well-known structural characterisation that the median of a tree is either a single vertex or two adjacent vertices. We also presented Algorithm 4.3, which runs in linear time to compute the distance-sums of all vertices in a tree. The median can then be located by one additional traversal through the vertices and their computed distance-sums.

The structural and algorithmic results on the medians of unweighted trees have direct generalisations to edge-weighted trees, stated by Theorem 4.14 and Algorithm 4.4. We also studied the medians of vertex-weighted trees, mainly in order to establish the lemmas required in Section 5.5.

We established the structural characterisation of the median of an unweighted tree using the fact that the distance-sum function is strictly convex. Nevertheless, despite the median of an edge-weighted tree has the same structural characterisation of the median of an unweighted tree, we needed to take a different approach because the distance-sum in an edge-weighted tree is *not* strictly convex. At the end of Section 4.2, we took a slight detour in Section 4.2.4 to complete the landscape by summarising the properties of convexity and

strict convexity of the eccentricity and the distance-sum in unweighted, edge-weighted, and vertex-weighted trees.

Future directions A possible next step is to investigate the centres and medians of k-trees, in both the structural and algorithmic aspects. Since k-trees are chordal (Proposition 2.15), the following proposition can be easily inferred.

Proposition 4.23. *The centre of a k-tree induces a biconnected subgraph that has diameter at most three but can have arbitrarily many vertices.*

Proof. Firstly, the centre of a chordal graph induces a biconnected subgraph (Chang [23]), so the same holds for k-trees. Figure 4.12 contains some 2-trees whose centres are biconnected but not 3-connected. This means that despite k-trees being more specific than chordal graphs, the connectivity of a k-tree's centre is still the same as the connectivity of a chordal graph's centre.

Similar to the previous paragraph, the centre of any chordal graph has diameter at most three (Chepoi [27]), so the same holds for k-trees. Furthermore, this is the strongest upper bound on the diameter of the centre of a 2-tree, and is realised by the 2-tree with diameter three in Figure 4.12(a).

Finally, for any constant $c \geq 4$, we show how to construct a 2-tree whose centre consists of c vertices. This construction starts with the 2-tree in Figure 4.12(b), and then adds $c-4$ vertices $v_1, \ldots, v_p$ such that every v_i is adjacent to u_1 and u_2. Then the centre of the final 2-tree is $\{u_1, u_2, w_1, w_2\} \cup \{v_1, \ldots, v_p\}$. As an example, Figure 4.12(c) is the 2-tree obtained by adding two vertices to $\{u_1, u_2\}$ in Figure 4.12(b). □

As a side note, the original leaf-removal algorithm for trees (Algorithm 4.1) does not directly work on 2-trees, as demonstrated by the counter-example in Figure 4.13. The centre of this 2-tree is $\{y_2, y_3, y_4\}$. By viewing 'leaves' as vertices of degree 2, Algorithm 4.1 sequentially deletes $\{x_8, y_8\}$, $\{x_7, y_7\}$ and so on. After $\{x_1, x_2\}$ is deleted on the last iteration, the set of remaining vertices $\{r_1, r_2\}$ is not the correct centre.

Moreover, the distance between the wrong output and the correct centre can be made arbitrarily large on a similar 2-tree with more vertices x_j's and y_j's, where each x_j is adjacent to $\{r_1, x_{j-1}\}$ and each y_j to $\{y_{j-2}, y_{j-1}\}$.

Nevertheless, there is no pressing need to develop a k-tree analogue of the leaf-removing Algorithm 4.1, as Granot and Zorin-Kapov [51] already presented a linear-time centre-locating algorithm for k-trees based on computing the eccentricity values. The eccentricity values of a k-tree's vertices can also be computed by the linear-time algorithm for chordal graphs, developed by Chepoi and Dragan [28].

The median of a k-tree seems harder to tackle than the centre, which is expected because the distance-sum is naturally more intricate than the eccentricity. This was demonstrated in the case of chordal graphs: Section 3.2 mentioned that the property of 'inducing a connected subgraph' is satisfied by a chordal graph's centre but not by its median. Although k-trees are more specific than chordal graphs and may satisfy stronger properties, some more sophisticated lemmas or tools seem necessary in order to characterise the structure of the medians of k-trees and to develop an efficient algorithm to locate them.

If new tools can be developed, one can then use these tools to attempt to develop centre-preserving and median-preserving simplifications of k-trees, and to prove their properties. Indeed, the properties and algorithms presented in this current chapter are ultimately to help with graph simplifications, the main motivation behind this thesis. On the other hand, one can study centrality concepts other than the centre and the median, and can focus on different graph classes—all these goals open up a wide range of possibilities.

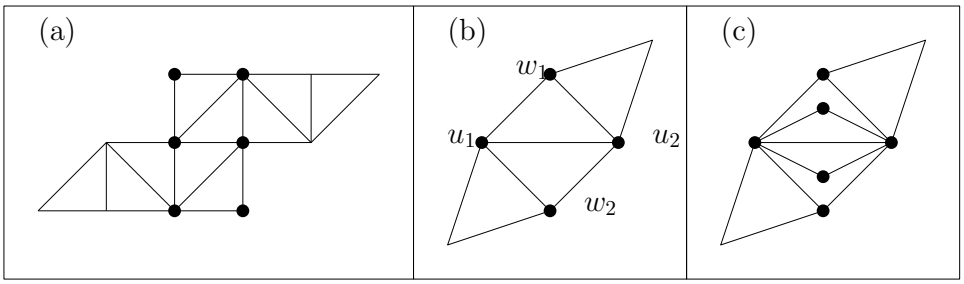


Figure 4.12: Some 2-trees and their centres.

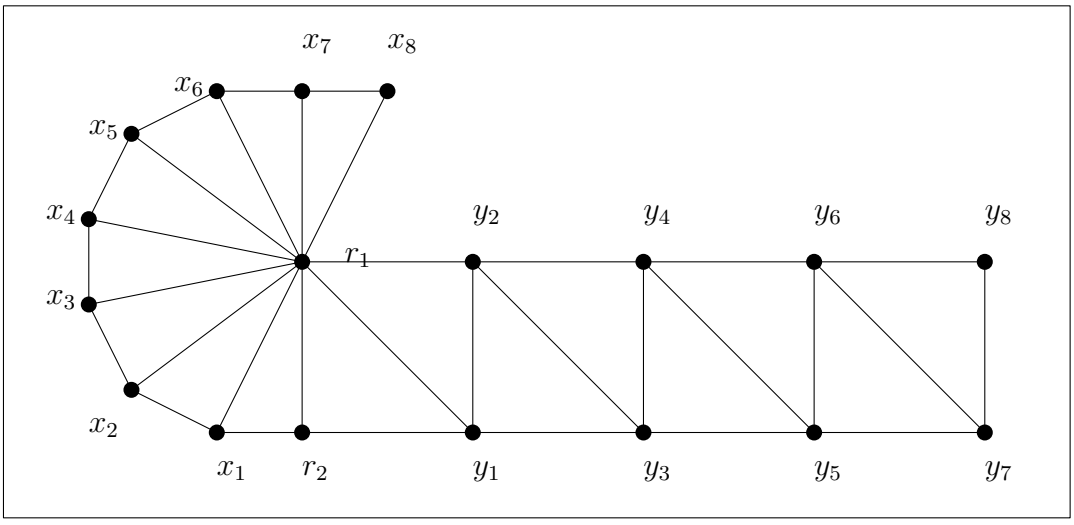


Figure 4.13: Leaf-removal for trees does not correctly locate the centres of 2-trees.

Chapter 5

Quasi-Isometric Vertex-Grouping

This chapter presents the main novel ideas of this thesis, which are related to graph simplification. We bring the concept of *quasi-isometries* from algebraic geometry into graph simplification, and use it to measure the distance distortion between the original graph and the simplified graph. This chapter begins with the definition and basic properties of quasi-isometries in Section 5.1.

In Section 5.2 we introduce a specific method of graph simplification. Given an input graph and a partition of its vertices into connected subsets, this simplification method constructs the so-called *partition-graph*. We then show that every partition-graph is quasi-isometric to the original graph.

Next, Section 5.3 discusses the novel concept called the *centre-shift*, which measures how much a graph simplification alters the location of the centre. If we have located the centre in the simplified graph, then a small centre-shift means that we can make good approximation on the location of the centre in the original graph.

We call a partition-graph of a tree a *partition-tree*. Section 5.4 studies the centre-shifts of partition-trees, and presents a specific procedure called *outward-contraction*, which produces partition-trees with zero centre-shift. In other words, outward-contraction preserves the centres in trees.

Finally, Section 5.5 studies partition-trees and their medians, and shows that every partition-tree preserves the median, provided that we store the number of vertices in each subset of the partition.

5.1 Quasi-Isometry

Many graph preprocessing methods have distance approximation as their main properties. Let G be the original graph and H be the simplified graph or the preprocessed structure. For every two vertices v_1 and v_2 in G, let $d_G(v_1, v_2)$ denote their distance in G, and we assume that their distance in H, denoted $d_H(v_1, v_2)$, is also well-defined. Now, examples of distance-approximating preprocessing methods from Section 3.1 include:

- A *spanner* of a graph G with constant-parameters $A \geq 1$ and $B \geq 0$ is a spanning subgraph such that for all vertices $v_1, v_2 \in G$:

$$d_H(v_1, v_2) \leq A \cdot d_G(v_1, v_2) + B.$$

- An *approximate distance oracle* of G with constant-parameter $A \geq 1$ is a data structure that efficiently returns an answer $d_H(v_1, v_2)$ to the distance query $d_G(v_1, v_2)$ for all vertices $v_1, v_2 \in G$, such that:

$$d_G(v_1, v_2) \leq d_H(v_1, v_2) \leq A \cdot d_G(v_1, v_2).$$

These inequalities have a similar form. The new distance function d_H is bounded above by the old distance function d_G, multiplied by a constant A, and then added by another constant B. This turns out to be captured by *quasi-isometry*, an established concept in Algebraic Geometry.

Quasi-isometries were first introduced by Gromov [52] to define the concept of *large-scale geometry*, which was then used to study infinite but finitely generated algebraic objects, such

as groups and their Cayley graphs. Later, quasi-isometries were also applied to infinite trees by Krön and Möller [73], and to infinite strings by Khoussainov and Takisaka [69].

Definition 5.1. Let M_1 and M_2 be two metric spaces, with d_1 and d_2 their respective metrics. Also, let A, B and C be non-negative integers with $A \geq 1$. Then a function $f : M_1 \to M_2$ is called a *quasi-isometry* with parameters A, B and C—an (A, B, C)-*quasi-isometry* for short—if the following two properties are satisfied.

(Q1) $\forall x, y \in M_1 \ : \ (1/A) \cdot d_1(x, y) - B \ \leq \ d_2\big(f(x), f(y)\big) \ \leq \ A \cdot d_1(x, y) + B.$

(Q2) $\forall y \in M_2 \ : \ \exists x \in M_1 \ : \ d_2\big(y, f(x)\big) \leq C.$

The first property (Q1) is called the *quasi-isometric inequality*, and the second property (Q2) is called the *density property*. The three parameters A, B and C are called the *quasi-isometric constants*. In particular, A is called the *stretch factor*, and B is called the *additive distortion*. This definition leads to the basic observations in the next proposition.

A related notion of distance-approximation is *bi-Lipschitz mappings*, which have the form

$$\frac{1}{A} \cdot d_1(x, y) \ \leq \ d_2\big(f(x), f(y)\big) \ \leq \ A \cdot d_1(x, y)$$

and play an important role in the field of metric embeddings (see Section 3.1.5). Hence quasi-isometries can be viewed as bi-Lipschitz mappings with additive distortions.

Proposition 5.2. *Every $(1, 0, 0)$-quasi-isometry is an isometry, every $(\alpha, \beta, 0)$-quasi-isometry is surjective, and every $(\alpha, 0, \gamma)$-quasi-isometry is injective.*

Proof. Recall the definition of isometries from the end of Section 2.2. Then $(1, 0, 0)$-quasi-isometry is an isometry simply by definition.

Next, when $C = 0$, the axiom (Q2) becomes

$$\forall y \in M_2 \ : \ \exists x \in M_1 \ : \ d_2\big(y, f(x)\big) = 0.$$

However, by the metric-space axiom (M1), $d_2(y, f(x)) = 0$ simply means that $y = f(x)$, which is precisely surjectivity.

Finally, to see that $B = 0$ implies injectivity, let $x, y \in M_1$ with $x \neq y$. Then axiom (M1) of M_1 ensures $0 < d_1(x, y)$. Since $A = \alpha \geq 1$ and $B = 0$, the inequality $0 < 1/\alpha \cdot d_1(x, y)$ also holds. Then (Q1) leads to

$$0 < \frac{1}{\alpha} \cdot d_1(x, y) < d_2\big(f(x), f(y)\big),$$

which means that $f(x) \neq f(y)$ due to axiom (M1) in M_2. Therefore f is injective. □

The most important property about quasi-isometries is that they form an equivalence relation. It is with this equivalence relation that the existing works by Gromov *et al.* partition some infinite class of algebraic objects into equivalence classes, and these classes are the so-called *large-scale geometries*. The rest of this section formally show that quasi-isometries form an equivalence relation.

Proposition 5.3. *Let $\mathcal{M}$ be a set of metric spaces, and define $\sim$ to be the binary relation on $\mathcal{M}$, such that two metric spaces $M_1 \sim M_2$ if and only if there is an (A, B, C)-quasi-isometry from M_1 to M_2 for some constants A, B and C.*

Then the relation $\sim$ is an equivalence relation.

Firstly, reflexivity follows directly from Definition 5.1. Every metric space is isometric to itself via the identity mapping, and such a mapping is a $(1, 0, 0)$-quasi-isometry as shown by Proposition 5.2. Establishing symmetry and transitivity requires more effort, so we separate these into the next two lemmas.

Lemma 5.4 (Symmetry of quasi-isometries)**.** *Let $f : M_1 \to M_2$ be an (A, B, C)-quasi-isometry. Then there exists an $(A, AB + 2AC, AB)$-quasi-isometry $\overline{f} : M_2 \to M_1$.*

Proof. For every $x \in M_2$, if x is mapped to by f, then set $\overline{f}$ to be any $v \in M_1$ such that $f(v) = x$. Otherwise, if x is not mapped to by f, then set $\overline{f}$ to be any $w \in M_1$ such that

$d_2(x, w) \leq C$, where the existance of such a w is guaranteed by the axiom (Q2).

Now we proceed to derive the quasi-isometry constants of $\overline{f}$. Consider any $x, y \in M_2$, and by the definition of $\overline{f}$:

$$d_2\big(x, f(\overline{f}(x))\big) \leq C \quad \text{and} \quad d_2\big(x, f(\overline{f}(x))\big). \tag{5.1}$$

Now, we first focus on one of the two inequalities ensured by (Q1) of f:

$$\begin{aligned}
\frac{1}{A} \cdot d_1\big(\overline{f}(x), \overline{f}(y)\big) - B &\leq d_2\big(f(\overline{f}(x), f(\overline{f}(y)\big) \\
&\leq d_2\big(f(\overline{f}(x), x\big) + d_2\big(x, y\big) + d_2\big(y, f(\overline{f}(y)\big) && \text{(by triangle ineq.)} \\
&\leq d_2\big(x, y\big) + 2C && \text{(by Inequalities (5.1)),}
\end{aligned}$$

and further re-arranging leads to:

$$d_1\big(\overline{f}(x), \overline{f}(y)\big) \leq A \cdot d_2\big(x, y\big) + A(B + 2C). \tag{5.2}$$

Next we consider the other inequality in (Q1):

$$\begin{aligned}
d_2\big(x, y\big) - 2C &\leq d_2\big(x, y\big) - d_2\big(f(\overline{f}(x), x\big) - d_2\big(f(\overline{f}(y), y\big) && \text{(by Inequalities (5.1))} \\
&\leq d_2\big(f(\overline{f}(x), f(\overline{f}(y)\big) && \text{(by triangle ineq.)} \\
&\leq A \cdot d_1\big(\overline{f}(x), \overline{f}(y)\big) + B && \text{(by (Q1) of } f\text{),}
\end{aligned}$$

and after re-arranging we have:

$$\frac{1}{A} \cdot d_2\big(x, y\big) - \frac{B + 2C}{A} \leq d_1\big(\overline{f}(x), \overline{f}(y)\big). \tag{5.3}$$

Now, with $A \geq 1$, observe that $-A(B + 2C) \leq -(B + 2C)/A$. Thus, Inequality 5.3 can be

re-written and combined with Inequality 5.2:

$$\frac{1}{A} \cdot d_2\big(x, y\big) - A(B + 2C) \quad \leq \quad d_1\big(\overline{f}(x), \overline{f}(y)\big) \quad \leq \quad A \cdot d_2\big(x, y\big) + A(B + 2C). \quad (5.4)$$

Finally, to derive the third constant of $\overline{f} : M_2 \to M_1$, consider any $x \in M_1$. The suitable choice of the element from M_2 turns out to be $f(x)$. Note that $f(\overline{f}(f(x))) = f(x)$ by the nature of the definition of $\overline{f}$. Now, after re-arranging (Q1) of f, we have:

$$\begin{aligned} d_1\big(x, \overline{f}(f(x))\big) &\leq A \cdot d_2\big(f(x), f(x)\big) + AB \\ &= A \cdot 0 + AB \\ &= AB. \end{aligned}$$

In conclusion, the quasi-isometric constants of $\overline{f} : M_2 \to M_1$ are respectively A, $A(B + 2C)$ and AB. □

Lemma 5.5 (Transitivity of quasi-isometries)**.** *Let $f : M_1 \to M_2$ be an (A_f, B_f, C_f)-quasi-isometry, and let $g : M_2 \to M_3$ be an (A_g, B_g, C_g)-quasi-isometry. Then there exists a quasi-isometry $h : M_1 \to M_3$ with constants $A = A_f A_g$, $B = A_g B_f + B_g$ and $C = A_g C_f + B_g + C_g$.*

Proof. Define a new mapping $h : M_1 \to M_3$ by setting $h(x) = g(f(x))$ for all $x \in M_1$. Now, for all $x, y \in M_1$, we have

$$d_2\big(f(x), f(y)\big) \leq A_f \cdot d_1\big(x, y\big) + B_f \quad \text{and}$$

$$d_3\big(g(f(x)), g(f(y))\big) \leq A_g \cdot d_2\big(f(x), f(y)\big) + B_g.$$

Substituting the first line into the second yields:

$$\begin{aligned} d_3\big(h(x), h(y)\big) &\leq A_g \cdot [A_f \cdot d_1\big(x, y\big) + B_f] + B_g \\ &= A_f A_g \cdot d_1\big(x, y\big) + (A_g B_f + B_g). \end{aligned}$$

Applying a similar argument for the lower bounds, we have

$$\frac{1}{A_f A_g} \cdot d_1\big(x, y\big) - \left(\frac{B_f}{A_g} + B_g\right).$$

With $A \geq 1$, observe that $B_f/A_g \leq A_g B_f$ and hence $-(A_g B_f + B_g) \leq -(B_f/A_g + B_g)$. Therefore,

$$\frac{1}{A_f A_g} \cdot d_1\big(x, y\big) - (A_g B_f + B_g) \quad \leq \quad d_3\big(h(x), h(y)\big) \quad \leq \quad (A_f A_g) \cdot d_1\big(x, y\big) + (A_g B_f + B_g).$$

As for the third quasi-isometry constant of h, note that for any $z \in M_3$, (Q2) of g ensures that there exists $y \in M_2$ with:

$$d_3\big(z, g(y)\big) \leq C_g. \tag{5.5}$$

By (Q2) of f applied to y, there exists $x \in M_1$ such that:

$$d_2\big(y, f(x)\big) \leq C_f. \tag{5.6}$$

Also, by (Q1) of g:

$$d_3\big(g(y), g(f(x))\big) \leq A_g \cdot d_2\big(y, f(x)\big) + B_g. \tag{5.7}$$

Finally, we combine Inequalities (5.5), (5.6) and (5.7), and derive that for any $z \in M_3$, there exists $x \in M_1$ such that $d_3\big(z, h(x)\big)$ is upper-bounded:

$$\begin{aligned}
d_3\big(z, h(x)\big) &\leq d_3\big(z, g(y)\big) + d_3\big(g(y), h(x)\big) && \text{(by triangle ineq.)} \\
&\leq C_g + d_3\big(g(y), h(x)\big) && \text{(by Inequality (5.5))} \\
&\leq C_g + A_g \cdot d_2\big(y, f(x)\big) + B_g && \text{(by Inequality (5.7))} \\
&\leq C_g + A_g C_f + B_g && \text{(by Inequality (5.6))}
\end{aligned}$$

In conclusion, $h : M_1 \to M_3$ is a quasi-isometry whose constants are $A_f A_g$, $A_g B_f + B_g$ and

$C_g + A_g C_f + B_g$ respectively. □

Now, the two Lemmas 5.4 and 5.5 above established the symmetry and transitivity of quasi-isometries, and completed the proof of Proposition 5.3, which states that quasi-isometries form an equivalence relation.

Note that every *finite* metric space with n elements is $(n, 1, n)$-quasi-isometric to the singleton space. This seems to suggest that quasi-isometries are only meaningful when applied to *infinite* spaces. However, the quasi-isometry constants need to get arbitrarily large in order to cover all finite spaces into one equivalence class. If we restrict the quasi-isometry constants to small values, then we can still let quasi-isometries map between finite spaces in a meaningful way.

Recall from the end of Section 2.2 that connected graphs can be viewed as metric spaces. This and the previous paragraph have shown that quasi-isometries (with small constants) are applicable to connected finite graphs, and from the next section we will employ quasi-isometries on the proposed graph simplification method called *partition-graphs*.

5.2 Partition-Graph

This section describes the generic graph simplification method based on partitioning the vertices into connected subsets.

Definition 5.6. A *partition* $\mathcal{P}$ of a graph G is a collection of subsets of $V(G)$ such that:

(a) every vertex in $V(G)$ belongs to exactly one subset in $\mathcal{P}$;

(b) every subset in $\mathcal{P}$ induces a connected subgraph.

The subsets in $\mathcal{P}$ are called *super-vertices*. Since every super-vertex X induces a connected subgraph, we can talk of the *diameter* of X.

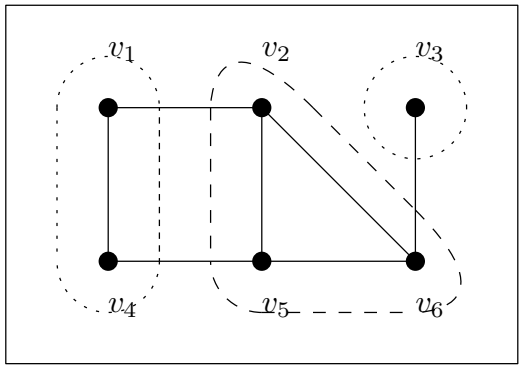


Figure 5.1: Invalid partition into subsets $\{v_1, v_4, v_3\}$ and $\{v_2, v_5, v_6\}$.

As a counter-example, consider the graph in Figure 5.1. The two subsets are $\{v_1, v_4, v_3\}$ and $\{v_2, v_5, v_6\}$. This is not a partition in the sense of Definition 5.6 because the subset $\{v_1, v_3, v_4\}$ does not induce a connected subgraph.

Based on the notion of partitions, we can now define the simplification method of *partition-graphs*.

Definition 5.7. Given a partition $\mathcal{P}$ on a graph G, the *partition-graph* $G(\mathcal{P})$ is defined as follows.

(a) The vertices of $G(\mathcal{P})$ are the super-vertices of $\mathcal{P}$.

(b) Two super-vertices X and Y are adjacent in $G(\mathcal{P})$ if and only if there exist $x \in X$ and $y \in Y$ such that $x \sim y$ in G.

Sometimes we write $\overline{G}$ instead of $G(\mathcal{P})$ if $\mathcal{P}$ is clear from the context. For a vertex $x \in G$, we use $\overline{x}$ to denote the super-vertex that contains x. Also, a super-vertex is sometimes regarded as a subset of G, and sometimes as a vertex of $\overline{G}$.

Figure 5.2 shows a partition, and $\{\overline{u}, \overline{v}, \overline{w}\}$ is the vertex-set of the simplified graph. Now, $\overline{u}$ and $\overline{v}$ are joined by a super-edge because $u_1 \sim v_1$ and $u_2 \sim v_2$. Similarly, $\overline{v}$ and $\overline{w}$ are joined by a super-edge because $v_2 \sim w_1$. However, $\overline{u}$ and $\overline{w}$ are not joined by a super-edge because there is no edge between any vertex in $\overline{u}$ and any vertex in $\overline{w}$. Therefore, the partition-graph induced by this partition is simply the three-vertex path $\overline{u} \sim \overline{v} \sim \overline{w}$.

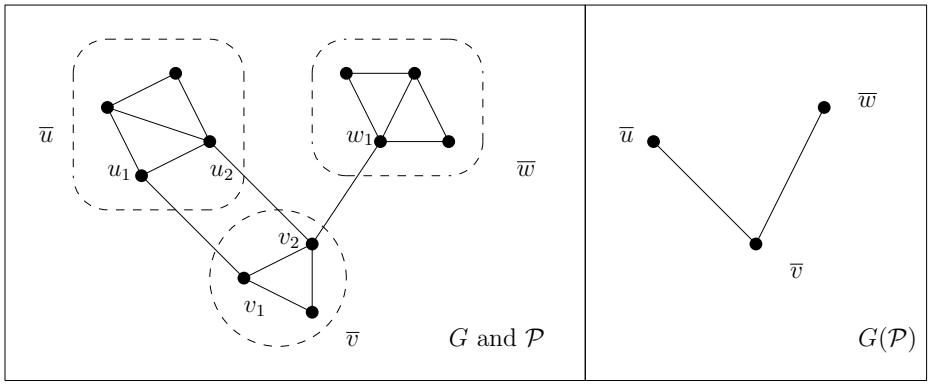


Figure 5.2: Example of a partition-graph.

A partition-graph is in fact a *graph minor*, which is the basis of a substantial theory on characterising graphs with the notion of *forbidden minors*, as covered by Chapter 13 in Diestel's classic textbook [35]. However, as this thesis is not related to the theory of forbidden minors, we only use the term 'partition-graphs'.

With a partition-graph, there is a natural mapping $\varphi : G \to \overline{G}$ with $\varphi(v) = \overline{v}$. In order for the mapping φ to be a quasi-isometry, we need an upper bound on the diameter of every super-vertex.

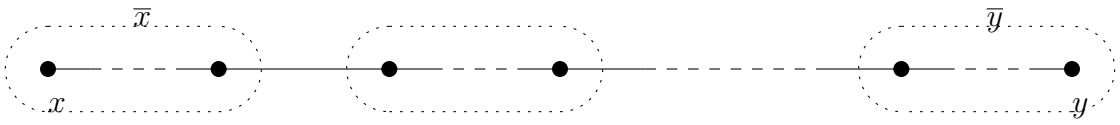
Definition 5.8. Given a constant $c \in \mathbb{N}_0$, a partition is called a *c-sharp partition* when every super-vertex $\overline{v}$ satisfies $\operatorname{diam}(\overline{v}) \leq c$.

For example, the partition $\mathcal{P}$ in Figure 5.2 is a 2-sharp partition; this is because the super-vertices $\overline{u}$ and $\overline{w}$ have diameter two, while the super-vertex $\overline{v}$ has diameter one.

An upper bound on the diameter of every super-vertex is akin to chopping the original graph into small and 'sharp' bits, and hence the term. Next, sharp partitions lead us to the following theorem related to quasi-isometry.

Theorem 5.9. *If $\mathcal{P}$ is a c-sharp partition on G, then the natural mapping φ from G to $H = G(\mathcal{P})$ is a quasi-isometry with constants $A = c + 1$, $B = 1$ and $C = 0$.*

Proof. Take any two vertices x and y in G, and consider their corresponding super-vertices $\overline{x}$ and $\overline{y}$. Firstly, observe that $d_H(\overline{x}, \overline{y}) \leq d_G(x, y)$. Next, given $d_H(\overline{x}, \overline{y})$, we seek the largest possible value of $d_G(x, y)$, which is achieved when the following occurs:



In this diagram, there are $d_H(\overline{x}, \overline{y})$ super-vertices on the path between x and y, so there are at most $(c+1) \cdot d_H(\overline{x}, \overline{y}) + c$ edges between x and y. This is the maximal value for $d_G(x, y)$, so

$$d_G(x, y) \leq (c+1) \cdot d_H(\overline{x}, \overline{y}) + c,$$

and after rearranging,

$$\frac{1}{c+1} \cdot d_G(x, y) - \frac{c}{c+1} \leq d_H(\overline{x}, \overline{y}).$$

Note that $c/(c+1) < 1$, so we can further simplify this inequality to

$$\frac{1}{c+1} \cdot d_G(x, y) - 1 \leq d_H(\overline{x}, \overline{y}).$$

With the lower bound established, we turn to the upper bound. Earlier we observed that $d_H(\overline{x}, \overline{y}) \leq d_G(x, y)$, but this can be relaxed to

$$d_H(\overline{x}, \overline{y}) \leq (c+1) \cdot d_G(x, y) + 1$$

because $c + 1 > 1$ and $1 > 0$. Therefore,

$$\frac{1}{c+1} \cdot d_G(x, y) - 1 \leq d_H(\overline{x}, \overline{y}) \leq (c+1) \cdot d_G(x, y) + 1,$$

which is the quasi-isometry inequality with $A = c + 1$ and $B = 1$. Now only the density property remains. For every super-vertex $X \in \overline{G}$, take any $x \in X$, and then $d_H\big(X, \varphi(x)\big) = d_H(X, X) = 0$. Therefore, the constant $C = 0$.

In conclusion, the function φ that maps each vertex x to $\overline{x}$ is a quasi-isometry with constants $A = c + 1$, $B = 1$ and $C = 0$. □

The sharpness of a partition ensures more precision with the distance. Indeed, the 0-sharp partition produces the original graph without any alteration, and the distance remains exactly preserved. However, such a partition does not effectively simplify the graph. In order to achieve a meaningful amount of simplification on top of approximate distance preservation, we also need the concept of *coarseness*.

Definition 5.10. Given a constant $b \in \mathbb{N}_0$, a partition is called a *b-coarse partition* when every super-vertex $\overline{v}$ satisfies $\operatorname{diam}(\overline{v}) \geq b$.

For example, the partition $\mathcal{P}$ in Figure 5.2 is a 1-coarse partition; this is because both $\overline{u}$ and $\overline{w}$ have diameter two, while $\overline{v}$ has diameter one.

Note that a lower bound on a super-vertex's diameter implies a lower bound on its cardinality. Namely, if $\operatorname{diam}(\overline{v}) \geq b$, then $\overline{v}$ must contain at least $b+1$ vertices.

The b-coarseness ensures that the size of the partition-graph $G(\mathcal{P})$ is $b+1$ times smaller than the size of the original graph G, as stated in the following proposition.

Proposition 5.11. *If $\mathcal{P}$ is a b-coarse partition on G, then*

$$|G(\mathcal{P})| \leq \frac{1}{b+1} \cdot |G|.$$

□

In conclusion, sharpness ensures distance precision and coarseness ensures effective simplification. A good partition must achieve a balance between these two antithetical parameters. Finally, we end this section with several path-related properties preserved by the partition-graph.

Proposition 5.12. *Let $[v_1, v_2, \ldots, v_p]$ be a path in G. Then the corresponding path $[\overline{v}_1, \overline{v}_2, \ldots, \overline{v}_p]$ in $\overline{G}$ is always shorter.*

A cycle is just a path that starts and ends at the same vertex, so the proposition above

also applies to cycles. Specifically, if G does not contain any cycle, then it is impossible for $\overline{G}$ to contain a cycle; this implies the corollary below.

Corollary 5.13. *Given any partition on a tree, the induced partition-graph is always a tree.*

5.3 Centre-Shift

When G is a graph whose centre C_G is impractical to compute, one might want to simplify G to a smaller graph H with a surjective mapping $\varphi : G \to H$, locate the centre of H (denoted C_H), and then use φ and C_H to infer C_G. The natural way to infer C_G is to use the reverse image:

$$\varphi^{-1}(C_H) = \{v \in G \mid \varphi(v) \in C_H\}.$$

This is the set of vertices in G that correspond to the centre in H. This set does not necessarily coincide with the original C_G, so we need some form of a metric to measure the distance between $\varphi^{-1}(C_H)$ and C_G. Hence, we introduce the concept of the centre-shift to quantify the distance in G between the subsets $\varphi^{-1}(C_H)$ and C_G.

Since H is a simplified and less detailed graph, defining the centre-shift in terms of the distance in G appears more accurate and reasonable. Also, note that φ has to be surjective in order for the following definition to make sense. Nonetheless, as H is viewed as a simplified version of G, it is reasonable for φ to be surjective. Indeed, the mapping defined for partition-graphs in Theorem 5.9 is a surjection.

Definition 5.14. The *centre-shift* of φ is defined to be $d_G\big(C_G, \varphi^{-1}(C_H)\big)$.

Before investigating any possible relationship between quasi-isometry and the centre-shift, we first present a lemma that relates the eccentricity with quasi-isometries. Given a quasi-isometry, the distance functions of the two graphs satisfy the quasi-isometric inequality of Definition 5.1. It turns out that the eccentricity-functions satisfy a similar inequality too.

Lemma 5.15. *Let G and H be graphs, with d_G and d_H their respective distance functions, and ecc_G and ecc_H their respective eccentricity-functions.*

If $\varphi : G \to H$ be a quasi-isometry with constants $A = \alpha$, $B = \beta$ and $C = \gamma$, then for every vertex $v \in G$,

$$\frac{1}{\alpha} \cdot \mathrm{ecc}_G(v) - \beta \leq \mathrm{ecc}_H\big(\varphi(v)\big) \leq \alpha \cdot \mathrm{ecc}_G(v) + \beta.$$

Proof. First, let $v' \in G$ be an ecc-wit of v. Then as $\mathrm{ecc}_G(v) = d_G(v, v')$,

$$\begin{aligned} \frac{1}{\alpha} \cdot \mathrm{ecc}_G(v) - \beta &= \frac{1}{\alpha} \cdot d_G(v, v') - \beta \\ &\leq d_H\big(\varphi(v), \varphi(v')\big) \\ &\leq \mathrm{ecc}_H\big(\varphi(v)\big). \end{aligned}$$

On the other hand, let $u \in G$ such that $\varphi(u)$ is an ecc-wit of $\varphi(v)$. Then

$$\begin{aligned} \mathrm{ecc}_H\big(\varphi(v)\big) &= d_H\big(\varphi(v), \varphi(u)\big) \\ &\leq \alpha \cdot d_G(v, u) + \beta \\ &\leq \alpha \cdot \mathrm{ecc}_G(v) + \beta. \end{aligned}$$

Combining these two inequalities completes the proof. □

The next two theorems derive upper bounds on the centre-shift, using the quasi-isometry inequality. These two theorems require the graph G to possess an extra condition called uniform eccentricity, as defined below.

Definition 5.16. Let G be a graph with centre C_G. Then G is said to have *uniform eccentricity* (*uni-ecc*) when it satisfies

$$\forall v \in G : d(C_G, v) = \mathrm{ecc}(v) - \mathrm{rad}(G).$$

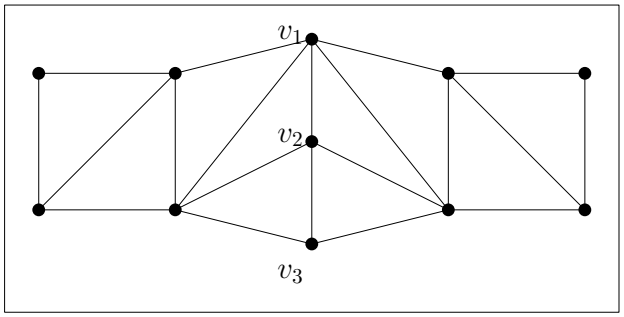


Figure 5.3: Example of a graph without the uni-ecc property.

Recall that the radius is simply the eccentricity of the centre-vertex. Thus, in other words, this property states that the eccentricity of a vertex v is the sum of the radius and distance between v and the centre.

Earlier, Proposition 2.2 observed that the difference between the eccentricities of any two adjacent vertices is no more than one. Put in the context of the definition above, the essence of Proposition 2.2 can also be expressed as

$$\forall v \in G : d(C_G, v) \geq \text{ecc}(v) - \text{rad}(G).$$

Therefore, the uni-ecc property is in fact a strong property. Intuitively, uni-ecc states that the eccentricities *always increase* step by step as we move layer by layer away from the centre. This is not satisfied in every graph, as shown by the counter-example in Figure 5.3. This is a chordal graph, in which v_1 is the sole centre-vertex and the radius is 2. Uni-ecc is not satisfied because $d(C_G, v_3) = d(v_1, v_3) = 2$ while $\text{ecc}(v_3) - \text{rad}(G) = 3 - 2 = 1$.

Now we begin to study if a quasi-isometry implies any upper bound on the centre-shift. The following theorem establishes that a quasi-isometry does imply a weak upper bound. This bound depends on the radius of the simplified graph, and assumes that the original graph has the uni-ecc property.

Theorem 5.17. *Let G be a graph with the uni-ecc property, and let C_G be the centre of G. Also, let $\psi : G \to H$ be a surjective quasi-isometry with constants $A = \alpha$, $B = \beta$ and $C = \gamma$.*

Then the centre-shift is bounded above by

$$\left(\alpha - \frac{1}{\alpha}\right)\operatorname{rad}(H) + \alpha\beta + \frac{\beta}{\alpha}.$$

Proof. Let $g \in C_G$ and $v \in \psi^{-1}(C_H)$ be vertices such that $d_G(g, v)$ equals the centre-shift:

$$d_G(g, v) = d_G\big(C_G, \psi^{-1}(C_H)\big).$$

By Proposition 2.2, $d_G(g, v) \leq \operatorname{ecc}_G(v) - \operatorname{ecc}_G(g)$. Then we rearrange the inequalities in Lemma 5.15 for v:

$$\begin{aligned}\frac{1}{\alpha}\operatorname{ecc}_G(v) - \beta &\leq \operatorname{ecc}_H\big(\psi(v)\big)\\ \operatorname{ecc}_G(v) &\leq \alpha\big(\operatorname{ecc}_H\big(\psi(v)\big) + \beta\big).\end{aligned}$$

Similarly for g:

$$\begin{aligned}\operatorname{ecc}_H\big(\psi(g)\big) &\leq \alpha \cdot \operatorname{ecc}_G(g) + \beta\\ \frac{1}{\alpha}\big(\operatorname{ecc}_H\big(\psi(g)\big) - \beta\big) &\leq \operatorname{ecc}_G(g)\\ -\operatorname{ecc}_G(g) &\leq -\frac{1}{\alpha}\big(\operatorname{ecc}_H\big(\psi(g)\big) - \beta\big).\end{aligned}$$

Combining these two inequalities, the steps above can be summarised as

$$\begin{aligned}d_G\big(C_G, \psi^{-1}(C_H)\big) &= d_G(g, v)\\ &\leq \operatorname{ecc}_G(v) - \operatorname{ecc}_G(g) && \text{(by Proposition 2.2)}\\ &\leq \alpha\big(\operatorname{ecc}_H\big(\psi(v)\big) + \beta\big) - \frac{1}{\alpha}\big(\operatorname{ecc}_H\big(\psi(g)\big) - \beta\big) && \text{(by Lemma 5.15)}\\ &= \alpha \cdot \operatorname{ecc}_H\big(\psi(v)\big) - \frac{1}{\alpha} \cdot \operatorname{ecc}_H\big(\psi(g)\big) + \alpha\beta + \frac{\beta}{\alpha}.\end{aligned}$$

Our assumption $v \in \psi^{-1}(C_H)$ means that $\psi(v) \in C_H$, so $\operatorname{ecc}_H\big(\psi(v)\big) \leq \operatorname{ecc}_H\big(\psi(g)\big)$.

Incorporating this into our reasoning,

$$d_G\big(C_G, \psi^{-1}(C_H)\big) \leq \left(\alpha - \frac{1}{\alpha}\right) \cdot \mathrm{ecc}_H\big(\psi(v)\big) + \alpha\beta + \frac{\beta}{\alpha}.$$

Since $\mathrm{ecc}_H\big(\psi(v)\big) = \mathrm{rad}(H)$, the proof is complete. □

Quasi-isometries describe the distance approximation for general graph simplifications. Nevertheless, in partition-graphs, the distance functions satisfy a property more specific than a quasi-isometry. For all vertices v_1 and v_2, the distance $d_H(\overline{v}_1, \overline{v}_2)$ in a partition-graph is always shorter than the distance $d_G(v_1, v_2)$ in the original graph. Note that this is not the case with edge-removing simplifications such as spanners, where distances in the simplified graph increases. Consequently, the 'one-sided quasi-isometry' satisfied by partition-graphs yields a finer bound.

Theorem 5.18. *Let G be a graph with the uni-ecc property, and let C_G be the centre of G. If $\varphi : G \to H$ is a surjective mapping that satisfies*

$$\frac{1}{\alpha} \cdot d_G(x, y) - \beta \leq d_H\big(\varphi(x), \varphi(y)\big) \leq d_G(x, y),$$

then the centre-shift is bounded above by $(\alpha - 1) \cdot \mathrm{rad}(H) + \alpha\beta$.

Proof. First we adjust the reasoning for Lemma 5.15. Let $v \in G$, and w an ecc-wit of v. Then

$$\begin{aligned} \frac{1}{\alpha} \cdot \mathrm{ecc}_G(v) - \beta = \frac{1}{\alpha} \cdot d_G(v, w) - \beta \\ &\leq d_H\big(\varphi(v), \varphi(w)\big) \\ &\leq \mathrm{ecc}_H\big(\varphi(v)\big). \end{aligned}$$

After rearranging, we have

$$\mathrm{ecc}_G(v) \leq \alpha\big(\mathrm{ecc}_H\big(\varphi(v)\big) + \beta\big). \tag{5.8}$$

As for the other direction, let $\varphi(v')$ be an ecc-wit of $\varphi(v) \in G$. Then

$$\begin{aligned}\mathrm{ecc}_H\big(\varphi(v)\big) &= d_H\big(\varphi(v), \varphi(v')\big)\\ &\leq d_G(v, v')\\ &\leq \mathrm{ecc}_G(v),\end{aligned}$$

and rearranging leads to

$$-\mathrm{ecc}_G(v) \leq -\mathrm{ecc}_H\big(\varphi(v)\big). \tag{5.9}$$

Next we modify the reasoning for Theorem 5.17. Let $g \in C_G$ and $v \in \varphi^{-1}(C_H)$ such that $d_G(g, v) = \big(C_G, \psi^{-1}(C_H)\big)$. Then

$$\begin{aligned}d_G\big(C_G, \psi^{-1}(C_H)\big) &= d_G(g, v)\\ &\leq \mathrm{ecc}_G(v) - \mathrm{ecc}_G(g) && \text{(by Proposition 2.2)}\\ &\leq \alpha\big(\mathrm{ecc}_H\big(\varphi(v)\big) + \beta\big) - \mathrm{ecc}_H\big(\varphi(g)\big) && \text{(by Inequalities 5.8 and 5.9)}\\ &\leq \alpha\big(\mathrm{ecc}_H\big(\varphi(v)\big) + \beta\big) - \mathrm{ecc}_H\big(\varphi(v)\big) && \text{(by } \mathrm{ecc}_H\big(\varphi(g)\big) \geq \mathrm{ecc}_H\big(\varphi(v)\big)\text{)}\\ &= (\alpha - 1) \cdot \mathrm{ecc}_H\big(\varphi(v)\big) + \alpha\beta\\ &= (\alpha - 1) \cdot \mathrm{rad}(H) + \alpha\beta,\end{aligned}$$

which establishes the upper bound. □

So far, this section has established two general upper bounds on the centre-shift for arbitrary graphs, and from the next section onwards, we will seek stronger results by focusing on trees.

5.4 Outward-Contraction and Centres of Trees

This section focuses on partition-trees and their centre-shifts. In the previous section, Theorem 5.17 established a generic bound on the centre-shift for graphs that satisfy the uni-ecc property in Definition 5.16. This in fact applies to trees, as shown by the following theorem.

Theorem 5.19. *In a tree T with centre C_T, every $v \in T$ satisfies $d(C_T, v) = \mathrm{ecc}(v) - \mathrm{rad}(T)$.*

The proof of this theorem requires another lemma first.

Consider the tree in Figure 5.4, where the centre consists of just the vertex c. If we fix v_1, then w_2 is an ecc-wit of c, where the path $[v_1 \dots w_2]$ passes through c. On the other hand, if we consider v_2, then w_1 is an ecc-wit of c such that the path $[v_1 \dots w_1]$ passes through c. This is the intuition behind Lemma 5.20 below.

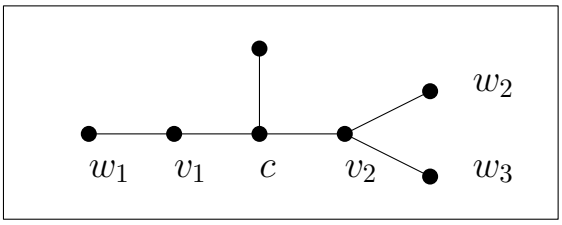


Figure 5.4: Example of Lemma 5.20.

Lemma 5.20. *Let v be any vertex in a tree T, and let $c \in C_T$ be the centre-vertex such that $d(v, c) = d(v, C_T)$. Then c has an ecc-wit w such that the path $[v \dots w]$ passes through c.*

Proof. A tree's centre consists of either one vertex or two adjacent vertices, so we can split the reasoning into two cases.

Suppose the centre consists of just one vertex c. Since c lies on the mid-point of a diameter-path, it must have at least two ecc-wits. Hence, no matter where our chosen v is, there always exists at least one ecc-wit w of c such that the path $[v \dots w]$ passes through c.

Next, suppose the centre consists of two adjacent vertices c_1 and c_2. Note that in this case, it is possible for a centre-vertex to have only one ecc-wit. Nonetheless, the path between

c_1 and any of its ecc-wits must pass through c_2, and *vice versa*. Hence, given any v, let c_1 be the centre-vertex that is closer to v without loss of generality. Then, the path between c_1 and every ecc-wit of c_1 must pass through c_2, so consequently the path between v and every ecc-wit of c_1 must pass through c_2. Therefore, there always exists at least one ecc-wit w of c_1 such that the path $[v \dots w]$ passes through c_1. □

Now, this lemma enables us to establish Theorem 5.19, which states that trees satisfy the uni-ecc property.

Proof of Theorem 5.19. For every vertex v, let c be its corresponding centre-vertex such that $d(v, c) = d(v, C_T)$. Then by Lemma 5.20, there is an ecc-wit w of c such that the path $[v \dots w]$ passes through c. This means that $d(v, w) = d(v, c) + d(c, w)$.

We now show that w is not only an ecc-wit of c, but also an ecc-wit of v. To do this, we need to establish that $\forall x \in T : d(v, x) \leq d(v, w)$. We look at two cases for x.

Firstly, suppose the path $[v \dots x]$ passes through c. Then

$$\begin{aligned} d(v, x) &= d(v, c) + d(c, x) \\ &\leq d(v, c) + d(c, w) && (w \text{ is ecc-wit of } c) \\ &= d(v, w). \end{aligned}$$

Secondly, suppose $[v \dots x]$ passes through c. Then consider the two paths $[v \dots c]$ and $[v \dots x]$. Let v' be the last common vertex on these paths. Now the tree can be pictured as in Figure 5.5.

With this, we can derive the following:

$$\begin{aligned} d(v,x) &= d(v,v') + d(v',x) \\ &= d(v,v') + d(c,x) - d(c,v') \\ &\leq d(v,v') + d(c,w) - d(c,v') && (w \text{ is ecc-wit of } c) \\ &< d(v,v') + d(c,w) + d(c,v') && (\text{adding positive } d(c,v') \text{ twice}) \\ &= d(v,w). \end{aligned}$$

Combining the two cases, we can conclude that w is an ecc-wit of v. This means that

$$\begin{aligned} \operatorname{ecc}(v) - \operatorname{rad}(T) &= \operatorname{ecc}(v) - \operatorname{ecc}(c) \\ &= d(v,w) - d(c,w) \\ &= d(v,c) \\ &= d(v,C_T), \end{aligned}$$

which is the statement of Theorem 5.19. □

Since trees satisfy the uni-ecc property, Theorems 5.17 and 5.18 are applicable to trees, and hence the centre-shift of any partition-tree is bounded by a linear function of the radius.

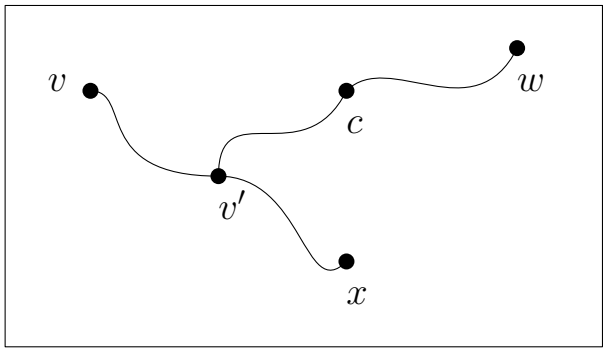


Figure 5.5: For the proof of Theorem 5.19.

Even so, the value of the centre-shift can still be arbitrarily large, as demonstrated by Figure 5.6. In each of the three trees, the original centre C_G consists of the solid vertices, the dotted lines indicate a 3-sharp partition, and the centre of the partition-tree is the super-vertex V. Then the centre-shift is $d(C_G, V)$. Hence, from top to bottom in Figure 5.6, the respective centre-shift is one, two and three.

If we extend this pattern by putting k 'triples' on the top branch and k 'pairs' on the bottom, for any arbitrary $k \geq 3$, then we can obtain a tree and a partition whose centre-shift is arbitrarily large. However, a large tree constructed in this way also has large radius, so the centre-shift increases together with the radius, which agrees with Theorem 5.18.

This example demonstrates that even in trees, not all partitions preserve the centre. Nevertheless, there is a specific method to construct a partition of a tree, such that the resulting partition-graph is guaranteed to have centre-shift zero. This method, called *outward-contraction*, is the subject of the rest of this section. Outward-contraction (Algorithm 5.5) takes a tree as the input, designates an arbitrary initial vertex,[1] and then builds the super-vertices by 'contracting' neighbourhoods from the initial vertex outwards.

Algorithm 5.5 Outward-Contraction

```
1: procedure OUTWARD(Any unrooted tree T)
2:     Root T at an arbitrary vertex r
3:     for For every vertex y with even lev(y) do
4:         Define the outward-neighbours of y as the set {v | v ~ y, lev(v) > lev(y)}
5:         Build a super-vertex consisting of y and its outward-neighbours
6:     end for
7: end procedure
```

Now it directly follows that outward-contraction always produces a 2-sharp and 0-coarse partition. Figure 5.7 shows several partitions produced by outward-contraction on a single tree, where the results are different depending on the choice of the initial vertex.

In Figure 5.7(a), v_1 is chosen as the initial vertex. As v_1 is on level zero, it is grouped with its only outward-neighbour v_2. Next, v_3 on the second level is grouped with v_4, and v_5

[1] Recall the notion of the *level*-function presented by Definition 2.5.

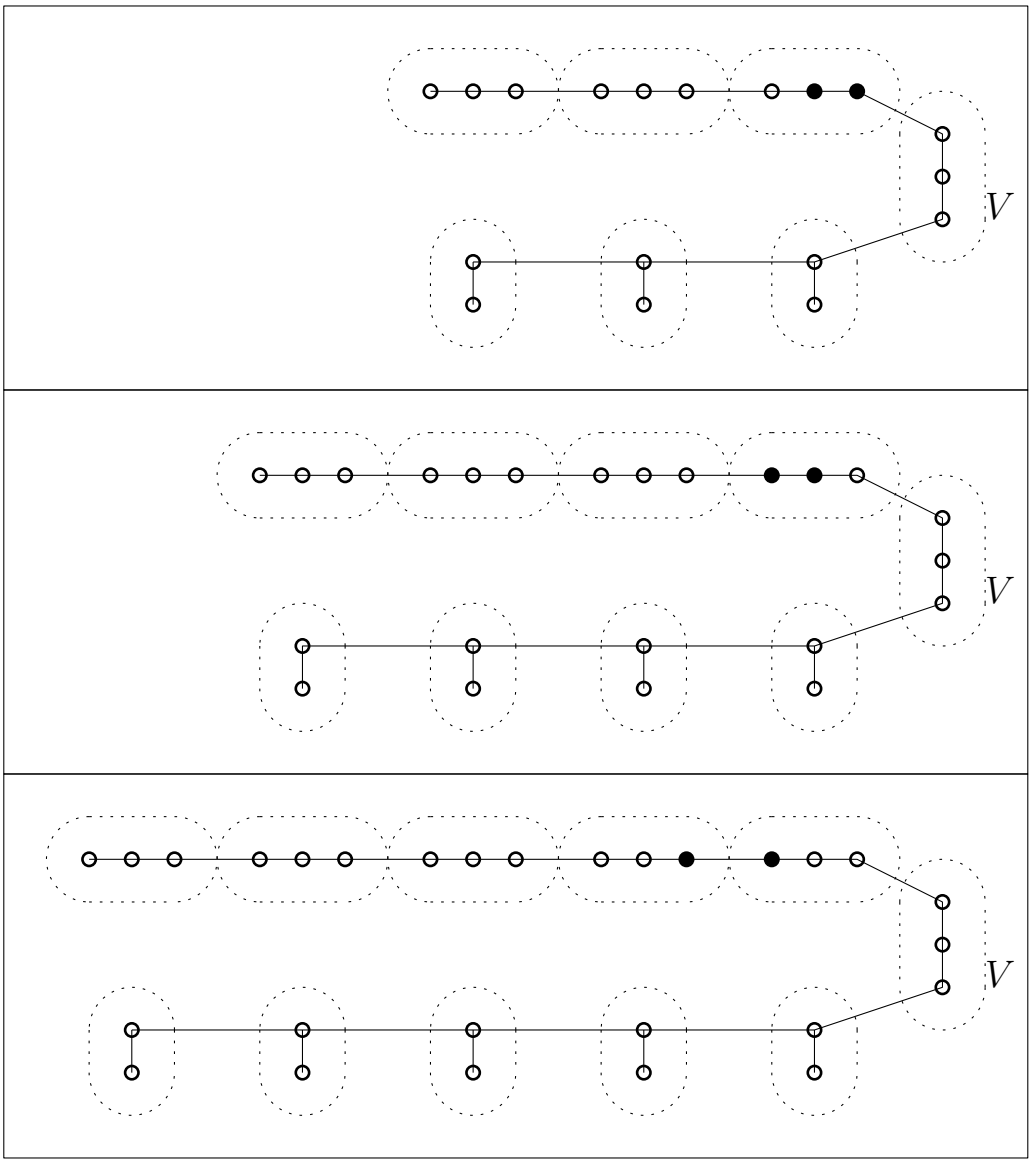


Figure 5.6: Pattern of partition-trees with centre-shift value arbitrarily large.

on the fourth level is grouped with v_6. Finally, the sixth level contains the remaining five vertices $\{v_7, u_1, u_2, u_3, u_4\}$, each of which becomes a super-vertex on its own.

In Figure 5.7(b), the initial vertex is v_2, and v_2 is grouped with its two outward-neighbours v_1 and v_3. On the second level, v_4 is grouped with its only outward-neighbour v_5. On the fourth level, v_6 is grouped with its outward-neighbours $\{u_1, u_2, u_3, u_4, v_7\}$.

In Figure 5.7(c), the initial vertex is v_6, which is grouped with its outward-neighbours and v_2 is grouped with its two outward-neighbours v_1 and v_3. On the second level, v_4 is grouped with its only outward-neighbour v_5. On the fourth level, v_6 is grouped with its outward-neighbours $\{u_1, u_2, u_3, u_4, v_7\}$.

Note that in these three examples, explicit calculation shows that the centre-shifts are all zero. In fact, as the rest of this section shows, outward-contraction always produces a partition-tree with centre-shift zero. To establish this result, we first need to reduce the centre-location problem from a tree to a path. This is possible by Theorem 4.3, which states that the centre of a tree always lies on a diameter-path, and hence is the same as the centre of any diameter-path of the tree.

On the path-graph P_n on n vertices, a partition can be expressed as a sequence of natural numbers that represent the sizes of the super-vertices from left to right. The numbers in such a sequence sum to n, so sequences like these are simply integer compositions. It turns out that the centre-shift of a partition on P_n can be conveniently calculated from this representation with integer compositions, so we make a slight digression to derive some useful tools.

An integer composition w of length k can be viewed as a path-graph P_k, with each number of w being a vertex. Since an integer composition represents a partition of P_n, its corresponding path-graph P_k is a partition-graph of P_n, where $n = w[1] + w[2] + \cdots w[k]$. Thus we call P_k a *partition-path* of P_n. Being a tree, the path-graph P_k has one or two centre-vertices. Correspondingly we can think of the integer composition w as having a centre.

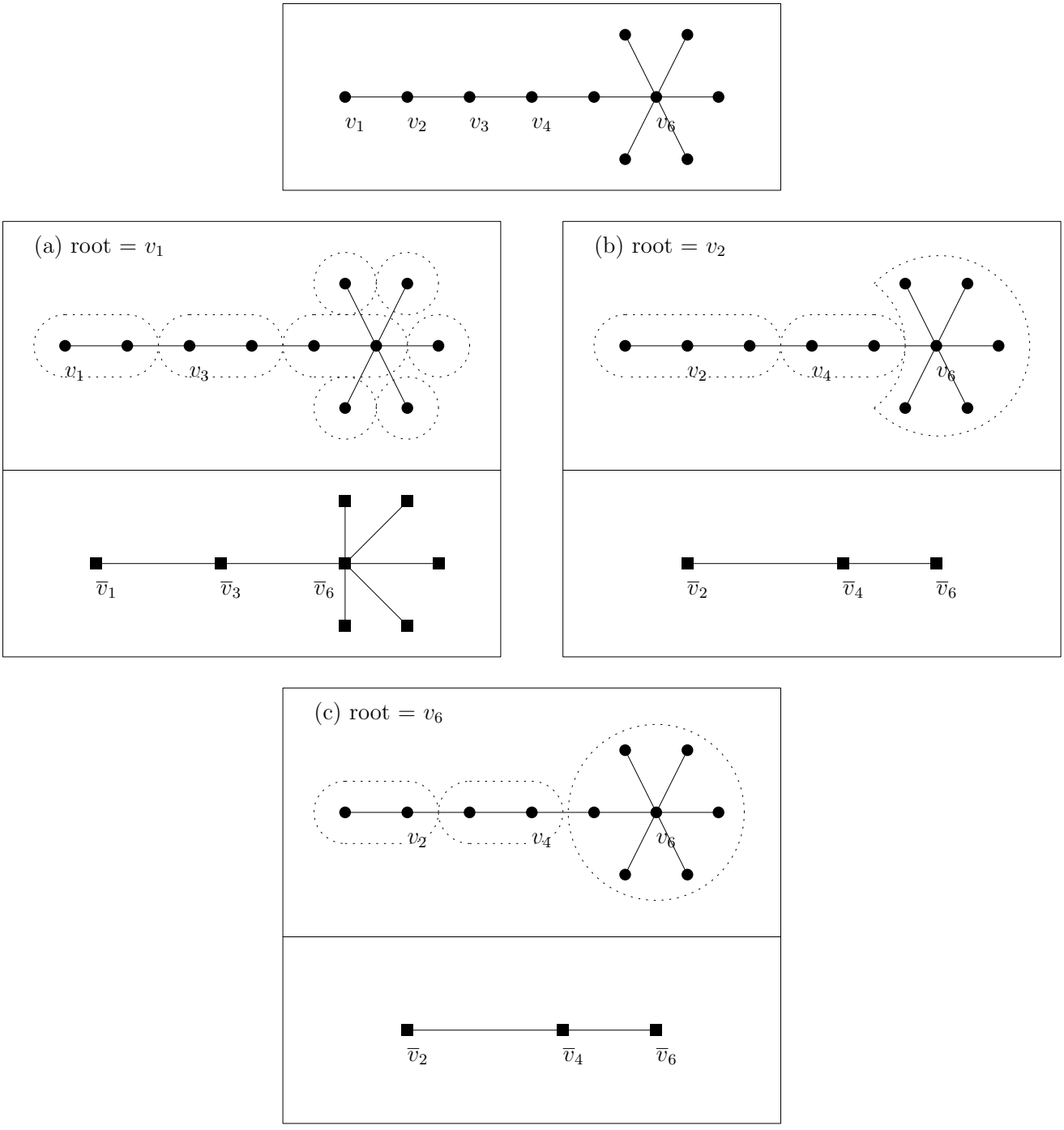


Figure 5.7: Partitions generated by outward-contraction on a tree.

Definition 5.21. Let w be an integer composition of length k. The set of *centre-indices* is $\{(k+1)/2\}$ when k is odd, and $\{k/2\ ,\ k/2+1\}$ when k is even. Furthermore,

- The *centre-sum* σ is $\sum w[i]$, for i in the set of centre-indices.
- The *left-sum* λ is $\sum w[i]$, for i smaller than all the centre-indices.
- The *right-sum* ρ is $\sum w[i]$, for i larger than all the centre-indices.

For example, consider $w =$ `332231`, which represents a partition on P_{14}. The centre-indices of w are 3 and 4, so the centre-sum $\sigma = w[3] + w[4] = 2 + 2 = 4$. Its left-sum is $\lambda = 6$, its right-sum is $\rho = 4$, and $|\lambda - \rho| = 2$. Now, since $\sigma \geq |\lambda - \rho|$, we can straightaway conclude that the centre-shift is zero. In general we have the following result with regards to the centre-shift.

Lemma 5.22. *Let P_k be a partition-path of P_n, and let w be the integer composition that represents P_k. Also, let σ, λ and ρ respectively denote the centre-sum, left-sum and right-sum of w. Then the centre-shift is*

$$\begin{cases} 0 & \text{if } \sigma \geq |\lambda - \rho| \\ \lceil \big(|\lambda - \rho| - \sigma\big)/2 \rceil & \text{otherwise.} \end{cases}$$

Proof. Let P_n be a path-graph with centre C_n, and let P_k be a partition-path of P_n with centre C_k. We picture P_n as Figure 5.8(a). Suppose $\varphi^{-1}(C_k)$ corresponds to Segment C on P_n, and has σ vertices. Then, let Segments L and R be the two shorter paths after removing Segment C from P_n. Suppose Segments L and R respectively contain λ and ρ vertices, and assume $\lambda \leq \rho$ without loss of generality.

Now we use the leaf-removal algorithm to locate the centre of P_n, and then calculate its distance to Segment C. On a path, one iteration of leaf-removal is the same as removing both endpoints. Hence, we first carry out λ iterations, which lead us to Figure 5.8(b). The

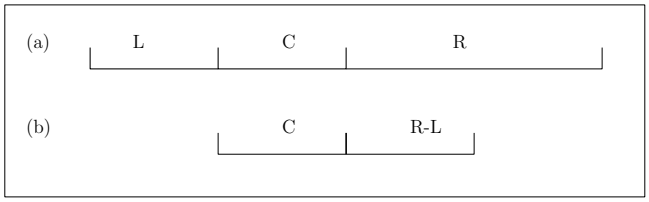


Figure 5.8: Locating the centre of the partition-path.

centre of this shorter path is exactly the same as the centre of P_n. From (b) there are three possible cases.

[Case 1] When Segments C and R-L have equal length, the centre of P_n is made up of the right-most vertex in Segment C and the left-most vertex in Segment R-L, so the centre-shift is zero.

[Case 2] When Segment C is longer than R-L, the centre of P_n lies in Segment C, so the centre-shift is zero.

These two cases above combine to prove the first part of the lemma: the centre-shift is 0 when $\sigma \geq |\lambda - \rho|$. In contrast, the final case involves more effort to quantify the non-zero centre-shift.

[Case 3] When Segment R-L is longer than C, the centre of P_n falls in Segment R-L, and the non-zero centre-shift is the distance between C_n and Segment C. This distance is the same as the distance between the right-most vertex in Segment C and the left-most vertex of C_n.

The right-most vertex in Segment C has index σ. On the other hand, on a path of length n, the index of the left-most centre-vertex is $\lceil n/2 \rceil$, so the left-most vertex of C_n has index

$$\left\lceil \frac{\sigma + |\lambda - \rho|}{2} \right\rceil .$$

Finally, we can derive the centre-shift by making the following subtraction:

$$\begin{aligned}\left\lceil \frac{\sigma + |\lambda - \rho|}{2} \right\rceil - \sigma &= \left\lceil \frac{\sigma + |\lambda - \rho|}{2} - \sigma \right\rceil \\ &= \left\lceil \frac{|\lambda - \rho| - \sigma}{2} \right\rceil,\end{aligned}$$

and this completes the second part of the theorem. □

Now we return to trees and outward-contraction.

Theorem 5.23. *Outward-contraction always produces a partition-tree with centre-shift zero.*

Proof. Let T be a tree, and let $\mathcal{P}$ be a partition produced by outward-contraction on T. Consider a diameter-path D in T. Let $\mathcal{P}_D$ be the restriction of $\mathcal{P}$ to D; that is, $\mathcal{P}_D = \{X \cap D \mid X \in \mathcal{P}\}$. In the rest of the proof, we focus only on the super-vertices in $\mathcal{P}_D$. Since D is a path, we refer to the sizes of its super-vertices as elements of an integer composition.

Earlier, Proposition 2.8 showed that every path in a rooted tree has at most one turning-point. If D does have a turning-point, then the super-vertex containing the turning-point has size either one or three. In the integer composition that represents $\mathcal{P}_D$, such a super-vertex is represented by a `1` or a `3`. On the other hand, the endpoints of D are contained in super-vertices with size one or two, and hence a `1` or a `2` in the integer composition. Meanwhile, all the remaining super-vertices that contain neither the turning-point nor endpoints always have size two. We now consider leaf-removal on D, which leads to two possible cases.

[Case 1] Suppose leaf-removal does not encounter the turning-point throughout the execution. This occurs when D has no turning-point, or when the super-vertex containing the turning-point is in the centre of $\overline{D}$ (the partition-path of D induced by $\mathcal{P}_D$). The centre of $\overline{D}$ contains either one or two super-vertices, and at most one of these super-vertices contains the turning-point.

If the centre of $\overline{D}$ contains only one super-vertex, then either this super-vertex contains the turning-point and has size one or three, or it does not contain a turning point and has

size two. Overall, the centre-sum of $\mathcal{P}_D$ is one, two or three.

If the centre of $\overline{D}$ contains two super-vertices, then the these super-vertices correspond to the following possible integer compositions: 12, 21, 32, 23 or 22. The first four occur when one of these super-vertices in the centre contains the turning-point, while the last composition 22 occurs when neither super-vertex in the centre contains the turning-point.

Now, the possible centre-sum σ ranges from one to five. Then there are four further subcases depending on whether each endpoint of D is a 1 or a 2. These subcases and their respective $|\lambda - \rho|$ values are listed in the table below. The centre-super-vertices are marked by [], and the dots all stand for 2.

subcase	$\lvert\lambda - \rho\rvert$	subcase	$\lvert\lambda - \rho\rvert$
12...[]...21	0	12...[]...22	1
22...[]...22	0	22...[]...21	1

As $\sigma \geq |\lambda - \rho|$ in all possible cases, by Lemma 5.22 the centre-shift is always zero.

[Case 2] Suppose leaf-removal encounters the turning-point of D at some point during the execution. Then the turning-point is not in any super-vertex of the centre of $\overline{D}$, so the possible values of the centre-sum are two (one super-vertex in the centre) and four (two super-vertices in the centre).

Without loss of generality, assume that the super-vertex containing the turning-point is on the left-hand side of the centre of $\overline{D}$. Depending on whether each endpoint is a 1 or a 2, as well as whether the super-vertex containing the turning-point is a 1 or a 3, there are eight subcases listed alongside the corresponding $|\lambda - \rho|$ values in the table below. Again, the super-vertices in the centre of $\overline{D}$ are marked by [], and the dots all stand for 2.

subcase	$\lvert\lambda - \rho\rvert$	subcase	$\lvert\lambda - \rho\rvert$
12..1..[].....21	1	12..3..[].....21	1
22..1..[].....22	1	22..3..[].....22	1
12..1..[].....22	2	12..3..[].....22	0
22..1..[].....21	0	22..3..[].....21	2

As $\sigma \geq |\lambda - \rho|$ in all cases, by Lemma 5.22, the centre-shift is always zero. □

5.5 Weighted-Partitions and Medians of Trees

Outward-contraction always produces centre-preserving partition-trees, but does not always preserve the median of a tree. In Figure 5.7, where the original median is $\{v_6\}$, outward-contraction starting from v_2 or v_6 does not preserve the median. Despite this, if the super-vertices' sizes are taken into account, the median of a tree can still be located from any partition-tree. For example, if we store the vertex-weights $(2, 2, 7)$ of the partition-tree in Figure 5.7(c), and compute the vertex-weighted version of the distance sum, then we obtain $\overline{v}_6$ as the median-part, which does correspond to the original median. This section extends the original notion of partition-graphs by storing the cardinalities of each super-vertex, as formally outlined by the the definition below.

Definition 5.24. Given a partition on a graph G, the *vertex-weighted partition-graph* $\overline{G}$ is defined as follows.

(a) The super-vertices and edges of $\overline{G}$ are the same as in Definition 5.7.

(b) For each super-vertex X in $\overline{G}$, its weight $f(X)$ is its cardinality.

Then we can apply the vertex-weighted distance sum from Definition 4.16 on the super-vertices of the vertex-weighted partition-graph. For each super-vertex X, its vertex-weighted distance sum is

$$\mathrm{ds}^w(X) = \sum_{Y \in \overline{G}} d'(X, Y) \cdot f(Y),$$

where the superscript w stands for *weighted*, and $d'(X, Y)$ denotes the distance in $\overline{G}$ between the super-vertices X and Y.

The super-vertices in $\overline{G}$ are denoted using capital letters, and a super-vertex X's distance sum in $\overline{G}$ is denoted by $\mathrm{ds}^w(X)$. Since there is little chance of ambiguity, we overload the notation for convenience.

Theorem 5.25. *Let T be a tree, and let $\overline{T}$ denote the vertex-weighted partition-tree induced by any partition on T. Then, every super-vertex in the median of $\overline{T}$ contains a vertex in the median of T.*

Proof. Since the vertex-weighted $\overline{T}$ is still a tree (Corollary 5.13), its median is either a single super-vertex of two adjacent super-vertices (Corollary 4.20).

[Case 1] Let X be the sole median super-vertex. Then by definition, for every neighbour Y of X, $\mathrm{ds}^w(X) < \mathrm{ds}^w(Y)$. Let $\overline{S}_X$ denote the set of super-vertices that passes through X in order to reach Y, and let $\overline{S}_Y$ be the analogous counterpart. Then by Corollary 4.18(ii), $f(\overline{S}_X) > f(\overline{S}_Y)$.

Let $x \in X$ and $y \in Y$ such that x and y are adjacent in T, and define S_x and S_y as in Lemma 4.17. Now by the definition of vertex-weighted partitions, $f(\overline{S}_X) = |S_x|$ and $f(\overline{S}_Y) = |S_y|$. This means that $|S_x| > |S_y|$, and hence $\mathrm{ds}^w(x) < \mathrm{ds}^w(y)$ by Corollary 4.18(i). By Lemma 4.21, $\mathrm{ds}^w(x) < \mathrm{ds}^w(v)$, for every $v \in S_y$. Since every vertex not in X has a larger distance sum than some other vertex, the median-vertices of T must be inside X.

[Case 2] Let X and Y be the two adjacent median parts in $\overline{T}$, and $x \in X$ and $y \in Y$ be the corresponding adjacent vertices in T. In addition, define $\overline{S}_X$, $\overline{S}_Y$, s_x and s_y as before. Now $\mathrm{ds}^w(X) = \mathrm{ds}^w(Y)$ implies $f(\overline{S}_X) = f(\overline{S}_Y)$. This further means that $|S_x| = |S_y|$, and hence $\mathrm{ds}^w(x) = \mathrm{ds}^w(y)$. Finally, using Corollary 4.22, x and y are the two median-vertices of T. □

5.6 Conclusion and Outlook

In this chapter, we started with *quasi-isometries*, and stated their definition and their basic properties of reflexivity, symmetry and transitivity. We then defined the graph simplification method called *partition-graphs*, and introduced the notions of *c-sharp* and *b-coarse* partitions, with $b, c \in \mathbb{N}$. Given a c-sharp partition, we showed that the corresponding partition-graph is quasi-isometric to the original graph with small constants. Apart from the sharpness that

ensures distance approximation, we also commented that the coarseness is required to ensure effective simplification.

Next, we introduced a new general concept called the *centre-shift* to measure how much a graph simplification changes the centre. Specifically, let $\varphi : G \to H$ be a graph simplification mapping, and C_G and C_H the centres of G and H; then the centre-shift of φ is defined to be $d_G(C_G, \varphi^{-1}(C_H))$. We found that a quasi-isometry alone leads to an upper bound on the centre-shift, provided that the graph G satisfies the *uniform-eccentricity* property.

This upper bound on the centre-shift is a function involving the radius of H. This suggests that particular methods of constructing quasi-isometric simplifications are needed to make the centre-shift a small number that is independent of the radius. In Section 5.4, we introduced a specific method called *outward-contraction* that constructs 2-sharp partition-graphs, and showed that outward-contraction applied to trees always produces a partition-tree with centre-shift zero.

We then turned from trees' centres to trees' medians. In Section 5.5, we showed that if we store the cardinalities of every super-vertex in the partition-graph $\overline{T}$, then we can use the vertex-weighted median in $\overline{T}$ to infer the location of the median of the original tree T.

As for future directions, the most important next step is to study the algorithmic aspects of constructing partition-graphs and quasi-isometric graph simplifications in general. So far, outward-contraction is the only concrete algorithm, and one could potentially use a similar idea to develop a greedy algorithm for generating partitions on general graphs. It might be computationally inefficient to check every pair of super-vertices in order to add every super-edge, so one could explore the possibility of 'spanners of partition-graphs' in more detail.

Similar to how Sections 5.4 and 5.5 focused on trees, one could also explore partition-graphs or other simplification methods for specific graph classes such k-trees or chordal graphs.

Finally, one could study the relationship between quasi-isometries and functions other than the distance. Among the countless properties and parameters of graphs, notable examples include the minimum cut, the conductance (related to spectral sparsifiers and spectral clustering from Section 3.1) and core-periphery structure (used in Social Network Analysis).

Chapter 6

Vertex-Grouping on Paths

Given fixed bounds on the sharpness and coarseness, a same graph has multiple partitions, which then correspond to multiple partition-graphs. Thus, it is a natural goal to seek common properties of all possible partition-graphs of a given graph. However, such a goal is challenging even on trees, let alone on general graphs, so we begin with paths.

As observed in Section 5.4, every partition-graph on P_n can be viewed as an integer composition by sequentially listing the sizes of the super-vertices. The ith part in the composition is the size of the ith super-vertex.

The sharpness and coarseness bounds of a partition on a path translate to bounds on the part-sizes of the corresponding composition. When a partition is c-sharp, every super-vertex has diameter at most c; on a path, this means that every part of the corresponding composition has size at most c. On the other hand, when a partition is b-coarse, every part of the composition has size at least b. In the rest of this chapter, an S-composition of n will be called a *sum-n composition over S*.

Section 6.1 begins this chapter by studying the set of all 2-sharp and 1-coarse partitions on P_n. This translates to the set of sum-n compositions with part-sizes either 2 or 3. We derive a bivariate recurrence to count the number of 'balanced' compositions—the sum-n compositions whose left-hand and right-hand halves have difference x. The *balance* of

a composition (formally defined later) is related to the centre-shift of the corresponding partition-path.

Section 6.2 presents a probabilistic algorithm SIMPLE that generates a 2-sharp 1-coarse partition on any graph by repeatedly choosing an unassigned vertex at random and grouping it with its unassigned neighbours. We show that the possible outcomes of SIMPLE on P_n correspond to sum-n compositions over $\{1, 2, 3\}$ that avoid the regular expression $\mathtt{12}^*\mathtt{1}$.

The algorithm SIMPLE may produce super-vertices of size one, which do not lead to effective simplification. Therefore, Section 6.3 presents an extension called LEFT-AUG, a procedure to generate partitions specifically on paths. Then we show that the possible outcomes of LEFT-AUG on P_n correspond to sum-n compositions over $\{2, 3, 4, 5\}$ that belong to another regular language.

6.1 Part-Sizes Two or Three

We begin by studying 2-sharp 1-coarse partitions on paths. It is too trivial if all parts have the same size, but too complex if there are too many possible part-sizes. Moreover, parts of size one are not useful in the practical sense. Thus, the case with part-sizes two or three is a reasonable starting point.

Table 6.1: All sum-n compositions over $\{2, 3\}$ with $n \leq 8$.

n				
1	—			
2	2			
3	3			
4	22			
5	23	32		
6	222	33		
7	223	232	322	
8	2222	233	323	332

As an example, Table 6.1 lists all sum-n compositions over $\{2, 3\}$, for n up to eight. For $n \geq 4$, every sum-n composition starts with either a 2 or a 3. If the first part is 2, then

after deleting the first part, the 'tail' is a sum-$(n-2)$ composition. On the other hand, if the first part is 3, then the tail is a sum-$(n-3)$ composition. Therefore, all of the sum-n compositions over $\{2, 3\}$ can be listed by:

- prepending a 2 to every sum-$(n-2)$ composition;
- prepending a 3 to every sum-$(n-3)$ composition; and
- joining the two lists above.

This observation leads to the following recurrence.

Proposition 6.1. *Let $p(n)$ denote the number of sum-n compositions over $\{2, 3\}$. Then,*

- *$p(1) = 0$ and $p(2) = 1$ and $p(3) = 1$;*
- *for all $n \geq 4$, $p(n) = p(n-2) + p(n-3)$.*

We now head towards the first 'average' property—the average length of all sum-n compositions. For example when $n = 8$, there are four sum-8 compositions[1] 2222, 233, 323 and 332, which have lengths 4, 3, 3 and 3 respectively. Then the average length is simply $(4 + 3 + 3 + 3)/4 = 15/4$.

In the general case, the denominator is settled by Proposition 6.1. Next, to calculate the numerator, we need the additional sequence defined in the next proposition.

Proposition 6.2. *Let $t(n)$ denote the total number of digits in the full list of sum-n compositions over $\{2, 3\}$. Then,*

- *$t(1) = 0$ and $t(2) = 1$ and $t(3) = 1$;*
- *for all $n \geq 4$, $t(n) = p(n) + t(n-2) + t(n-3)$.*

Before presenting the proof, consider the full list of sum-8 compositions as an example. The left-hand side of the following table is the list of sum-8 compositions arranged vertically.

[1] See Table 6.1.

We then divide this array of numbers into three regions, indicated by the lines on the right-hand side of the table.

2222	2	222
233	2	33
323	3	23
332	3	32

Now, the number of digits in left-hand column is the same as the number of sum-8 compositions. Meanwhile, the top-right region is the full list of sum-6 compositions, and the bottom-right region is the full list of sum-5 compositions. This illustrates the proof of the recurrence of $t(n)$.

Proof of Proposition 6.2. We begin with the two base cases. First, $t(1) = 0$ because there is no sum-1 composition over $\{2, 3\}$, and hence no digits at all. Second, $t(2) = 1$ because there is exactly one sum-2 composition with length one, so the total number of digits is 1. The same reasoning applies to $t(3) = 1$.

For a fixed $n \geq 4$, list the sum-n compositions vertically, and place two lines in this array of digits. The first line separates the array into the first column and the remaining right-hand region. Then the second line divides the sum-n compositions into those starting with 2 and those starting with 3.

The number of digits in the first column is exactly the number of sum-n compositions, which is $p(n)$ in Proposition 6.1.

The top-right region consists of the tails of the sum-n compositions that start with 2, so they are exactly the full list of sum-$(n-2)$ compositions. Hence, the total number of digits in the top-right region is $t(n-2)$. Similarly, the total number of digits in the bottom-right region is $t(n-3)$.

Finally, the total number of digits in the list of sum-n compositions is the sum of $p(n)$, $t(n-2)$ and $t(n-3)$, which establishes the recurrence. □

Then, the average length can be calculated by taking the numerator from Proposition 6.2 and the denominator from Proposition 6.1, as stated in the following corollary.

Corollary 6.3. *Let $a(n)$ denote the average length of sum-n compositions over $\{2, 3\}$. Then $a(n) = t(n)/p(n)$.*

Next, we move to 'balance' properties of sum-n compositions, which are related to the centre-shift of the corresponding partition-path. Earlier, Lemma 5.22 showed that the centre-shift of a partition-path is based on the difference between the corresponding composition's left-hand and right-hand halves. When this difference is small, we can intuitively picture a sum-n composition as being 'balanced'. Apart from the application to the centre-shift, the balance of a composition may be of pure combinatorial interest too.

The rest of this section works towards a recurrence that counts the number of balanced compositions as a bivariate sequence $b(n, d)$, where n is the sum and d is the 'balance'. There turn out to be two slightly different notions of 'balance', which respectively correspond to the pure combinatorial setting and the centre-shift setting. These are presented in Definitions 6.4 and 6.5 below.

Definition 6.4. Let s be a sum-n composition of length k. Then the *plain balance* of s is defined to be the absolute value of the difference between:

- the sum of the left $\lfloor k/2 \rfloor$ digits and
- the sum of the right $\lfloor k/2 \rfloor$ digits.

For example, the plain balance of 222333 is 3. The length is six, so we add the left three digits 222, add the right three digits 333, and take the absolute difference:

$$|(2 + 2 + 2) - (3 + 3 + 3)| = |6 - 9| = 3.$$

Note that when the length is odd, the middle digit is omitted.

Table 6.2: The base cases of $b_p(n, d)$ and $b_g(n, d)$.

plain b_p			
	d		
n	0	1	2
1			
2	1		
3	1		
4	1		
5	0	2	
6	2		

gapped b_g			
	d		
n	0	1	2
1			
2	1		
3	1		
4	1		
5	2		
6	2		

The plain balance separates a sum-n composition into the intuitive left-hand and right-hand halves, and this is the natural definition to use in a pure combinatorial setting. Nevertheless, in the context of the centre-shift, we need to take the centre-indices[2] into account. In 222333 for example, the centre-shift setting does not include the middle 23 when computing the value $|\lambda - \rho|$, where λ and ρ are the left-sum and right-sum as in Definition 5.21. As a result, we need another notion of the 'balance'.

Definition 6.5. Let s be a sum-n composition of length k. Then the *gapped balance* of s is defined to be the absolute value of the difference between:

- the sum of the left $\lfloor k/2 \rfloor - 1$ digits and
- the sum of the right $\lfloor k/2 \rfloor - 1$ digits.

Using the same example 222333, the gapped balance is 2, which is obtained by $|(2+2) - (3+3)| = |4-6| = 2$. Note that if the length k is odd, the plain balance and gapped balance are the same.

Having defined the two balance parameters of compositions, we now count the numbers of sum-n compositions over $\{2, 3\}$ that have a given number $d \in \mathbb{N}_0$ as their plain or gapped balance. The recurrence for the plain balance and the recurrence for the gapped balance are stated in the next two theorems.

[2]See Definition 5.21.

Theorem 6.6. *For $d \in \mathbb{N}_0$, let $b_p(n, d)$ denote the number of sum-n compositions over $\{2, 3\}$ with plain balance d. Then for $n < 7$, the values of $b_p(n, d)$ are listed in Table 6.2, and for $n \geq 7$, $b_p(n, d)$ satisfy the following recurrence.*

- $b_p(n, 0) = b_p(n-4, 0) + b_p(n-5, 1) + b_p(n-6, 0)$
- $b_p(n, 1) = b_p(n-4, 1) + 2 \cdot b_p(n-5, 0) + b_p(n-5, 2) + b_p(n-6, 1)$
- *For $d \geq 2$,*

  $b_p(n, d) = b_p(n-4, d) + b_p(n-5, d-1) + b_p(n-5, d+1) + b_p(n-6, d)$

Proof. Firstly, the base cases of b_p are explicitly determined from the full lists of sum-n compositions[3] in Table 6.1. Then for the general case, we use the term *diff*-d to mean a plain balance of d. Thus, $b_p(n, d)$ is the number of diff-d sum-n compositions over $\{2, 3\}$, where the term diff-d in this proof means a plain balance of d.

Every composition has two endpoints, and after deleting these two endpoints, the resulting composition is called the *middle*. In this proof, we analyse the possible combinations of the endpoints and the resulting middle. Namely, to count the number of diff-d sum-n compositions, we enumerate the possible combinations of the endpoints, and for each subcase we recursively use the result of $b(n', d')$ where $n' < n$ and $d' < d$.

[Case 1: $b(n, 0)$] First, there are three ways to obtain a diff-0 sum-n composition:

(i) Start with a diff-0, sum-$(n-4)$ composition, and append a `2` on both ends.

(ii) Start with a diff-1, sum-$(n-4)$ composition, and append a `2` and a `3` to the two ends such that the resulting sum-n composition is diff-0.

If the left-hand half is one larger than the right-hand half, then the `2` is added to the left and the `3` is added to the right. On the other hand, if the right-hand is bigger than the left, then the `3` is added to the left and the `2` added to the right.

(iii) Start with a diff-0, sum-$(n-6)$ composition, and append a `3` on both ends.

[3] Since all compositions in this section are over $\{2, 3\}$, we shall omit the mention of the set $\{2, 3\}$.

Therefore,

$$b(n,0) = b(n-4,0) + b(n-5,1) + b(n-6,0).$$

[Case 2: $b(n,1)$] Then, to obtain a diff-1 sum-n composition:

(i) Start with a diff-1, sum-$(n-4)$ composition, and append a 2 on both ends.

(ii) Start with a diff-0, sum-$(n-5)$ composition, and append a 2 to the left and a 3 to the right.

(iii) Start with a diff-0, sum-$(n-5)$ composition, and append a 3 to the left and a 2 to the right.

(iv) Start with a diff-2, sum-$(n-5)$ composition, and append a 2 and a 3 on the two ends such that the result is a diff-1 sum-n composition. (Similar to (ii) of the previous case.)

(v) Start with a diff-1, sum-$(n-6)$ composition, and append a 3 on both ends.

Therefore, $b(n,1) = b(n-4,1) + 2 \cdot b(n-5,0) + b(n-5,2) + b(n-6,1)$. The second term has a factor of 2 because it covers both (ii) and (iii).

[Case 3: $b(n,d)$ for $d \geq 2$] Finally, to obtain a diff-d sum-n composition:

(i) Start with a diff-d, sum-$(n-4)$ composition, and append a 2 on both ends.

(ii) Start with a diff-$(d-1)$, sum-$(n-5)$ composition, and append a 2 and a 3 on the two ends such that the result is a diff-d composition on n.

This subcase should be contrasted with (ii) and (iii) in the previous case, where there are two ways of adding a 2 and a 3 to extend from diff-0 to diff-1. However, when $d \geq 2$, there is only one way of adding a 2 and a 3 to extend from diff-$(d-1)$ to diff-d.

(iii) Start with a diff-$(d+1)$, sum-$(n-5)$ composition, and append a 2 and a 3 on the two ends such that the result is a diff-d composition on n.

(iv) Start with a diff-d, sum-$(n-5)$ composition, and append a 3 on both ends.

Therefore, for $d \geq 2$, $b(n,d) = b(n-4,d) + b(n-5,d-1) + b(n-5,d+1) + b(n-6,d)$. □

The next theorem states the recurrence that counts the number of sum-n compositions over $\{2,3\}$ with gapped balance d, which turns out to be identical to the plain-balance counterpart except for the base cases.

Theorem 6.7. *For $d \in \mathbb{N}_0$, let $b_g(n,d)$ denote the number of sum-n compositions over $\{2,3\}$ with gapped balance d. Then for $n < 7$, the values of $b_g(n,d)$ are listed in Table 6.2, and for $n \geq 7$, $b_g(n,d)$ satisfy the following recurrence.*

- $b_g(n,0) = b_g(n-4,0) + b_g(n-5,1) + b_g(n-6,0)$
- $b_g(n,1) = b_g(n-4,1) + 2 \cdot b_g(n-5,0) + b_g(n-5,2) + b_g(n-6,1)$
- *For $d \geq 2$,*

  $b_g(n,d) = b_g(n-4,d) + b_g(n-5,d-1) + b_g(n-5,d+1) + b_g(n-6,d)$

Proof. The base cases of $b_g(n,d)$ for $n < 7$ are determined explicitly from the full list in Table 6.1, and reasoning for the general case is identical to the proof of Theorem 6.6, with a 'diff-d' composition meaning a composition with a gapped balance of d instead. Hence we omit the details. □

Given the recurrences in Theorems 6.6 and 6.7, we can easily compute the values of $b_p(n,d)$ and $b_g(n,d)$. In particular, the gapped balance is related to the centre-shift of partition-paths, and $b_g(n,0) + b_g(n,1) + b_g(n,2)$ is the number of 2-sharp 1-coarse partitions of P_n with centre-shift zero.

Potential future work includes deriving the generating functions, and analysing the asymptotic properties. Nevertheless, bivariate generating functions do not always have closed-form expressions, so this goal is a challenging one. Another potential future goal is to extend the notions of balance to other part-sizes; in this section we focused only on compositions over $\{2,3\}$, and it would be of interest to investigate the balance-properties of compositions with part-sizes restricted to $A \subset \mathbb{N}$ or with unrestricted part-sizes.

Algorithm 6.6 Simple Randomised Contraction

```
1: procedure SIMPLE(any graph G)
2:     while unassigned vertices remain do
3:         Uniformly choose an unassigned vertex v
4:         Create a new part, consisting of v and its unassigned neighbours
5:     end while
6: end procedure
```

6.2 Simple Randomised Contraction

The previous section concerns a hypothetical setting without specifying how the partitions are generated. This present section introduces a specific randomised vertex-grouping algorithm, and investigates its properties on paths. During the execution of this method, we call a vertex *assigned* when it has been grouped into some part, or *uassigned* otherwise. This method, called Simple Randomised Contraction (Algorithm 6.6, abbreviated SIMPLE), chooses a vertex from the unassigned vertices uniformly at random, builds a new part consisting of this starting vertex and all its unassigned neighbours, and repeats until every vertex is assigned. A vertex is called a *starting vertex* when it is chosen on Line 3 during the execution of Algorithm 6.6.

The first question about Algorithm 6.6 is the average number of super-vertices of all the possible outcomes. Every execution of Algorithm 6.6 is associated with a probability and an outcome partition, so the set of possible outcomes' cardinalities can be viewed as a random variable, and we seek its expectation. This question happens to coincide with the *discrete parking problem* discussed in Section 3.3, and this expectation is $\frac{1}{2} - \frac{1}{2e^2} \approx 0.43233$.

In a composition that corresponds to an outcome of SIMPLE, each part has size either one, two or three. However, not every sum-n composition over $\{1, 2, 3\}$ corresponds to an outcome of SIMPLE; for example, SIMPLE can never produce `1221` on P_6. In terms of formal languages, we now characterise the compositions that represent the possible outcomes of SIMPLE. Namely, the possible outcomes of SIMPLE on P_n can be represented by compositions over the alphabet $\{1, 2, 3\}$ with an additional constraint: the compositions must not contain

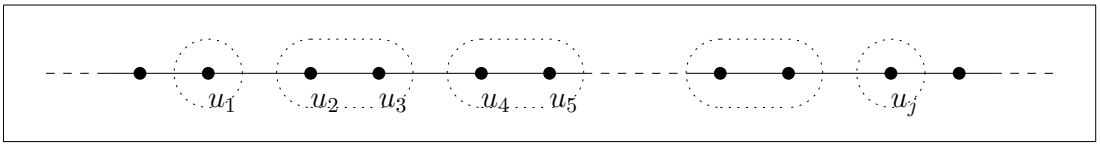


Figure 6.1: Forbidden pattern in the outcomes of SIMPLE.

substrings belonging to the set $\texttt{12}^*\texttt{1}$. Here, a *substring* means a consecutive subsequence in the composition, while the asterisk is the Kleene-star operation, such that $\texttt{12}^*\texttt{1}$ denotes the set $\{\texttt{11}, \texttt{121}, \texttt{1221}, \texttt{12221}, \ldots\}$.

Theorem 6.8. *A sum-n composition over* $\{1, 2, 3\}$ *corresponds to an outcome of* SIMPLE *on the n-vertex path if and only if it contains no substring from* $\texttt{12}^*\texttt{1}$.

Proof. ($\Rightarrow$) For a contradiction, consider a composition that is produced by SIMPLE but contains a substring from $\texttt{12}^*\texttt{1}$. Firstly, if the contained substring is $\texttt{11}$, then the partition has two neighbouring size-1 parts. This is impossible because one of these two vertices must be chosen first, and this firstly chosen vertex would absorb the other.

Next, if the composition contains a substring from $\texttt{12}^*\texttt{1}$, then the partition has the form shown by Figure 6.1. Let $u_1, u_2, \ldots, u_j$ denote the vertices that correspond to the occurrence of the substring from $\texttt{12}^*\texttt{1}$. Without loss of generality, we can assume u_1 is a starting vertex. This then implies that u_3 is also a starting vertex, which further leads to $u_5, u_7, \ldots, u_{j-1}$ all being starting vertices. Since u_{j-1} is a starting vertex, u_j cannot become a size-1 super-vertex on its own. This contradicts the assumption of a substring $\texttt{12}^*\texttt{1}$. Hence, If a composition corresponds to an outcome of SIMPLE, then it contains no substring from $\texttt{12}^*\texttt{1}$.

($\Leftarrow$) We use induction on the length of the composition. The base cases are the length-1 compositions $\texttt{1}$, $\texttt{2}$ and $\texttt{3}$. None of them contains a substring from $\texttt{12}^*\texttt{1}$, and we can explicitly check that each of them corresponds to a run of SIMPLE as follows. The composition $\texttt{1}$ represents the only possible partition on P_1. The composition $\texttt{2}$ represents the only possible partition on P_2. The composition $\texttt{3}$ represents the partition on P_3 where all three vertices are grouped into one super-vertex, which occurs when SIMPLE chooses the middle vertex in the first iteration.

Next, assume that every composition with no substring from $\texttt{12}^*\texttt{1}$ corresponds to at least one run of SIMPLE. A run of SIMPLE uniquely corresponds to a *starting-sequence*, which is a sequence of starting vertices chosen by SIMPLE during the execution.

For a composition w with no substring from $\texttt{12}^*\texttt{1}$, let k denote its length and s denote its sum, and assume that w corresponds to an outcome of SIMPLE on P_s. Now, consider adding an extra symbol $\texttt{x} \in \{1, 2, 3\}$ to the end of w to obtain a new composition w'. This new composition w' corresponds to a partition on $P_{s+\texttt{x}}$, and the inductive step needs to check whether this partition w' on $P_{s+\texttt{x}}$ corresponds to an outcome of SIMPLE, provided that w' contains no substring from $\texttt{12}^*\texttt{1}$.

The composition w can end in 1, 2 or 3, while the newly added symbol can be 1, 2 or 3, so there are nine possible cases in total, listed in the table below as well as in Figures 6.2, 6.3 and 6.4. The rest of the proof checks each case one by one. For each case, we assume that w is a composition that contains no substring from $\texttt{12}^*\texttt{1}$ and corresponds to an outcome of SIMPLE; this is the inductive hypothesis that will be used in most of the cases below.

end of w	extensions 1	2	3
..1	..11	..12	..13
..2	..21	..22	..23
..3	..31	..32	..33

The first three cases below are illustrated by the three blocks in Figure 6.2.

[Case 11] If w ends with 1 and is appended with an extra 1, then w' contains a substring from $\texttt{12}^*\texttt{1}$. Now that the premise is false, we do not need to check whether w' corresponds to an outcome of SIMPLE.

[Case 21] Suppose w ends with 2 and is appended with an extra 1. If w has the form $\texttt{..12}^*\texttt{2}$, then appending the 1 would make w' contain a substring from $\texttt{12}^*\texttt{1}$. Therefore,

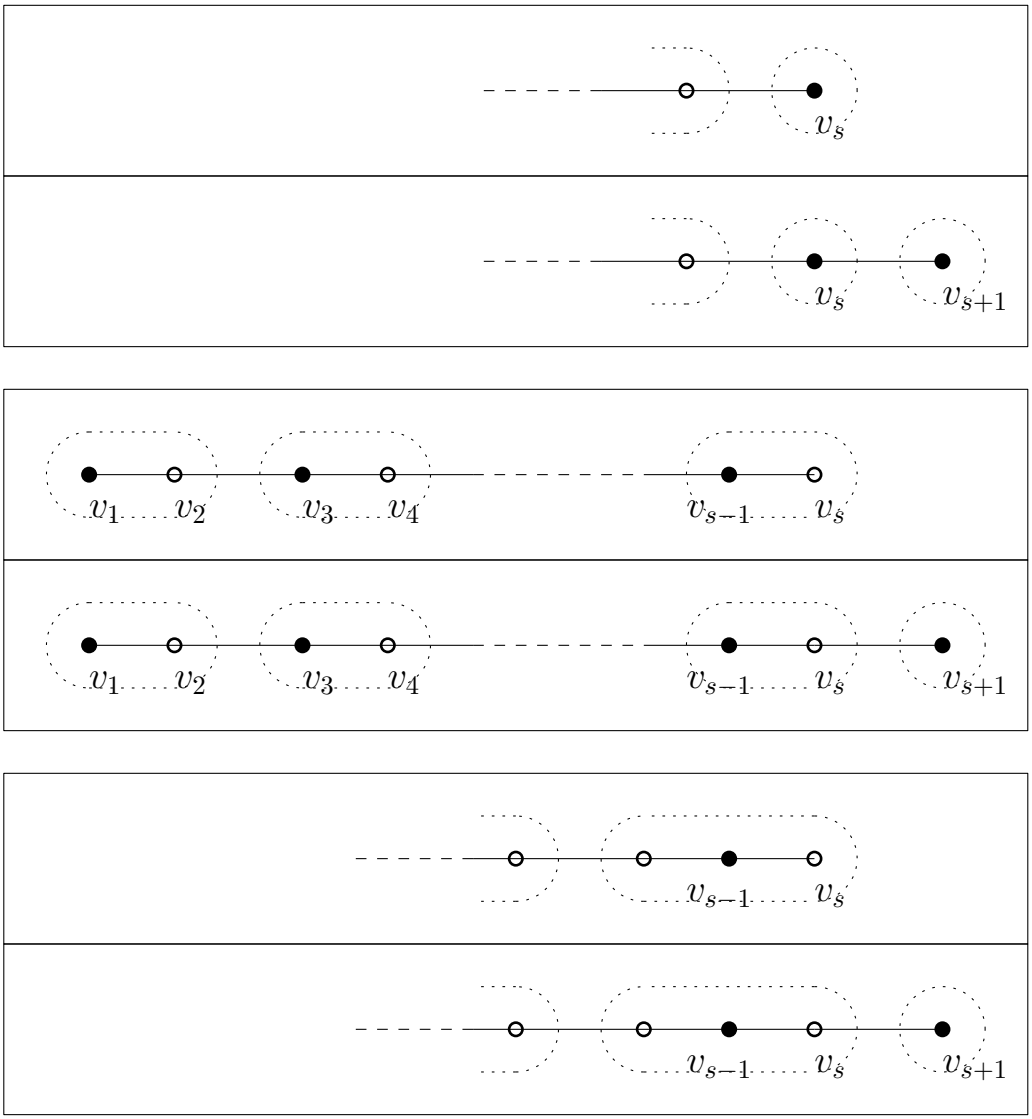


Figure 6.2: Inductive cases for SIMPLE, appending 1.

either w consists of 2s only, or w has the form $w_1 3 w_2$, where w_1 is another composition with no substring from 12^*1, and w_2 is a suffix with 2s only.

In the first scenario, the particular form of w enables us to infer a particular sequence. Among the possible runs of SIMPLE that produces w, the starting-sequence $Q = [v_1, v_3, \ldots, v_{s-1}]$ is one of them. (See the middle block of Figure 6.2.) Let Q' be the starting-sequence obtained by appending v_{s+1} to the end of Q. Then when SIMPLE follows Q', it produces the partition w' on P_{s+1}.

In the second scenario, let $j \in \mathbb{N}_0$ such that w_1 correspond to the vertices $\{v_1, \ldots, v_j\}$ and w_2 to $\{v_{j+4}, \ldots, v_s\}$. Since w_1 is a composition that contains no substring from 12^*1, by the inductive hypothesis, SIMPLE follows some starting-sequence Q_1 to produce the partition w_1 on P_j. Meanwhile, using the same reasoning from the previous paragraph, SIMPLE produces w_2 on P_{s-j-3} by following the starting-sequence $Q_2 = [v_{j+4}, v_{j+6}, \ldots, v_{s-1}]$. Now, let Q' be the concatenation of v_{j+2}, Q_1, Q_2 and v_{s+1}. Then when SIMPLE follows Q', it produces the partition w' on P_{s+1}. Therefore, w' corresponds to a run of SIMPLE.

[Case 31] If w ends with 3 and contains no substring from 12^*1, then appending with a 1 cannot create a substring in 12^*1. Next, by the inductive hypothesis, SIMPLE follows some starting-sequence Q to produce the partition w on P_s. Let Q' be the starting-sequence obtained by appending v_{s+1} to the end of Q. Then when SIMPLE follows Q', it produces the partition w' on P_{s+1}. Therefore, w' corresponds to a run of SIMPLE.

The next three cases are shown by the three blocks in Figure 6.3.

[Case 12] If w ends with 1 and contains no substring from 12^*1, then appending with a 2 cannot create a substring in 12^*1. Next, by the inductive hypothesis, SIMPLE follows some starting-sequence Q to produce the partition w on P_s. Let Q' be the starting-sequence obtained by inserting v_{s+2} before the occurrence of v_s in Q. Then when SIMPLE follows Q', it produces the partition w' on P_{s+2}. Therefore, w' corresponds to a run of SIMPLE.

[Case 22] If w ends with 2 and contains no substring from 12^*1, then appending with a 2

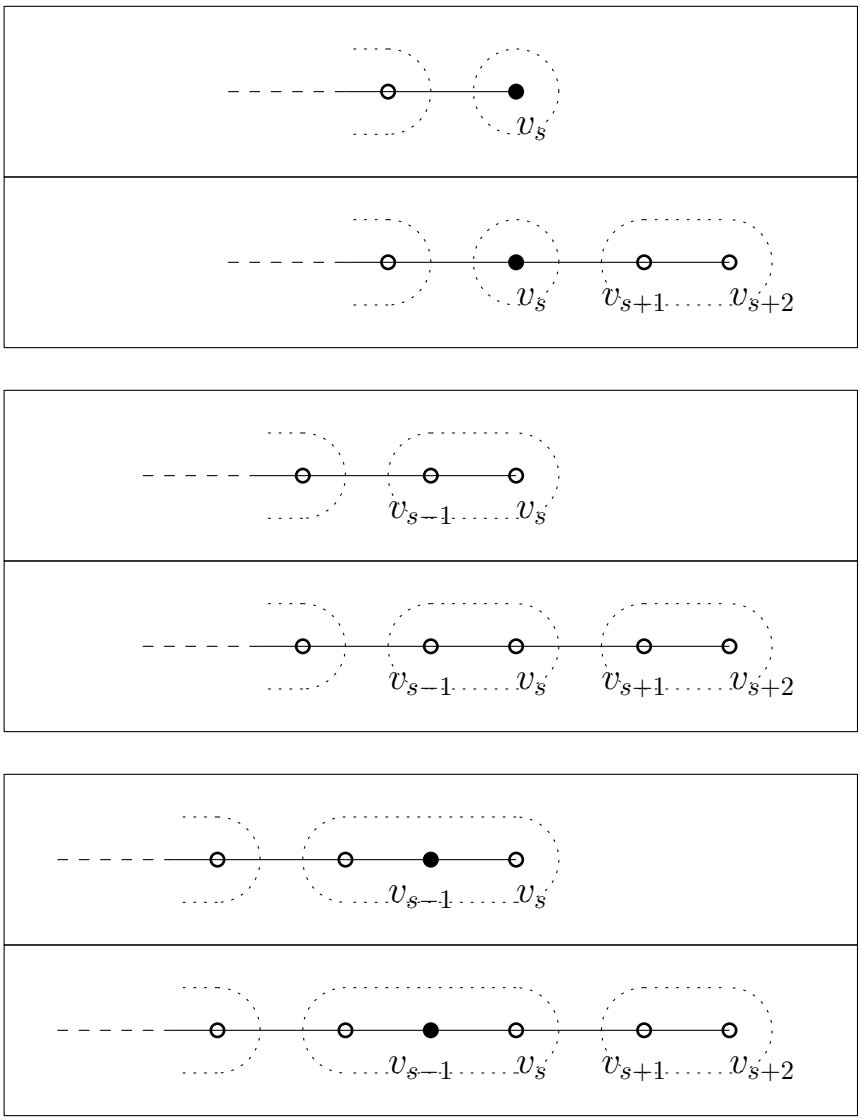


Figure 6.3: Inductive cases for SIMPLE, appending 2.

cannot create a substring in $\texttt{12}^*\texttt{1}$. Next, by the inductive hypothesis, SIMPLE follows some starting-sequence Q to produce the partition w on P_s. Since w ends with a part with size 2, Q must contain one of v_{s-1} or v_s.

If Q contains v_{s-1}, then we build the sequence Q' by appending Q with v_{s+1}. Now when SIMPLE follows Q', it produces the partition w' on P_{s+2}. If Q contains v_s, then we build the sequence Q' by inserting v_{v+2} before the occurrence of v_s in Q. Now when SIMPLE follows Q', it produces the partition w' on P_{s+2}.

Therefore, w' corresponds to a run of SIMPLE.

[Case 32] If w ends with `3` and contains no substring from $\texttt{12}^*\texttt{1}$, then appending with a `2` cannot create a substring in $\texttt{12}^*\texttt{1}$. Next, by the inductive hypothesis, SIMPLE follows some starting-sequence Q to produce the partition w on P_s. Let Q' be the starting-sequence obtained by appending v_{s+2} to the end of Q. Then when SIMPLE follows Q', it produces the partition w' on P_{s+2}. Therefore, w' corresponds to a run of SIMPLE.

The next three cases are shown by the three blocks in Figure 6.4.

[Case 13] If w ends with `1` and contains no substring from $\texttt{12}^*\texttt{1}$, then appending with a `1` cannot create a substring in $\texttt{12}^*\texttt{1}$. Next, by the inductive hypothesis, SIMPLE follows some starting-sequence Q to produce the partition w on P_s. As w ends with `1`, v_s must occur in Q. Let Q' be the starting-sequence obtained by inserting v_{s+2} before the occurrence of v_s in Q. Then when SIMPLE follows Q', it produces the partition w' on P_{s+3}. Therefore, w' corresponds to a run of SIMPLE.

[Case 23] If w ends with `2` and contains no substring from $\texttt{12}^*\texttt{1}$, then appending with a `3` cannot create a substring in $\texttt{12}^*\texttt{1}$. Next, by the inductive hypothesis, SIMPLE follows some starting-sequence Q to produce the partition w on P_s. As w ends with `2`, only one of v_{s-1} or v_s occurs in Q. Let Q' be the starting-sequence obtained by inserting v_{s+2} before the occurrence of v_{s-1} or v_s in Q. Then when SIMPLE follows Q', it produces the partition w' on P_{s+3}. Therefore, w' corresponds to a run of SIMPLE.

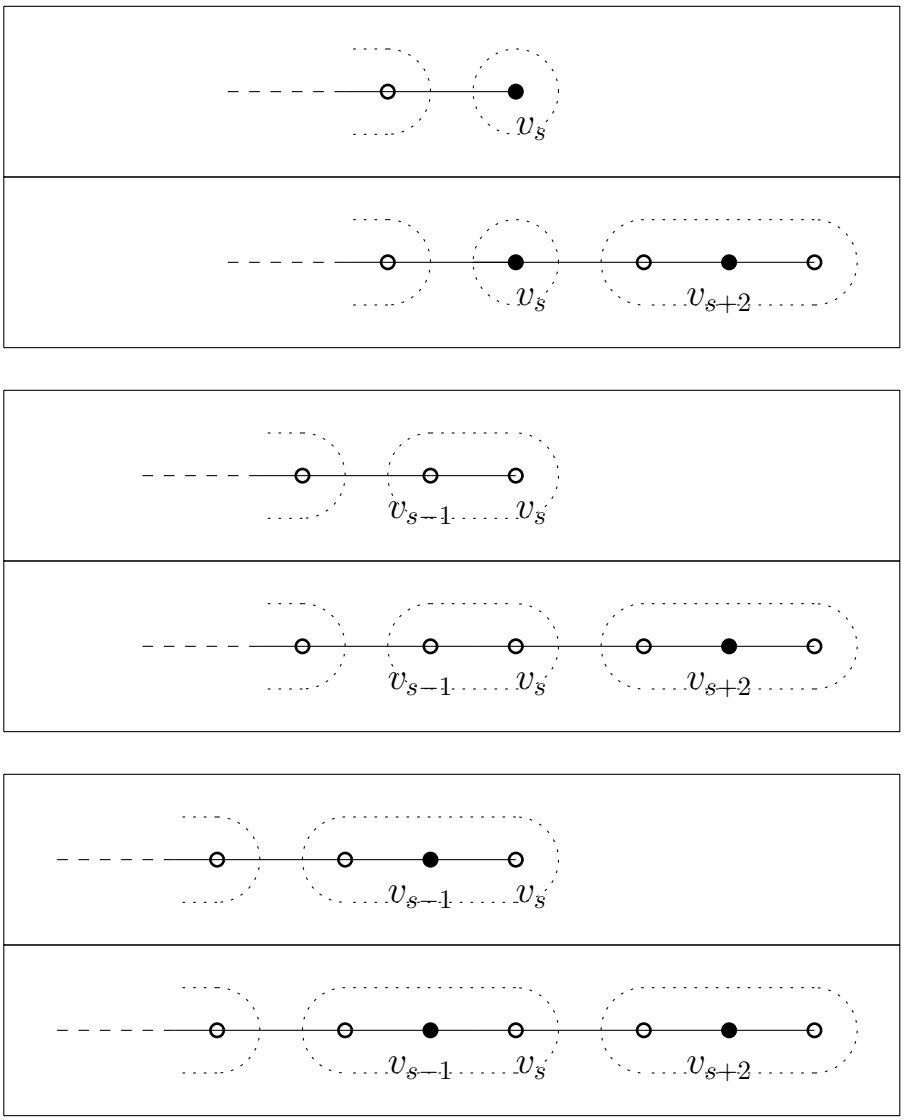


Figure 6.4: Inductive cases for SIMPLE, appending 3.

[Case 33] If w ends with `3` and contains no substring from `12*1`, then appending with a `3` cannot create a substring in `12*1`. Next, by the inductive hypothesis, SIMPLE follows some starting-sequence Q to produce the partition w on P_s. Let Q' be the starting-sequence obtained by appending Q with v_{s+2}. Then when SIMPLE follows Q', it produces the partition w' on P_{s+3}. Therefore, w' corresponds to a run of SIMPLE.

After routinely verifying all nine cases, we conclude that if a composition contains no substring from `12*1`, then it corresponds to an outcome of SIMPLE. □

Having characterised the SIMPLE compositions, we now turn to counting their numbers. Although the constraint with `12*1` makes a concise recurrence seem elusive, we can still bound the number of SIMPLE compositions using existing results from the theory of restricted integer compositions.

Every sum-n composition with letters either `2` or `3` contains no substring from `12*1`, so it corresponds to an outcome of SIMPLE by the characterisation theorem above. Hence, a lower bound of the number of sum-n SIMPLE composition is the number of sum-n compositions over $\{2, 3\}$, which is exactly the same sequence in Proposition 6.1.

On the other hand, every sum-n SIMPLE composition is itself a sum-n composition over $\{1, 2, 3\}$, so an upper bound is the number of sum-n compositions over $\{1, 2, 3\}$, whose recurrence relation can be derived in a similar way too.

We can also use a computer program to explicitly compute the number of SIMPLE compositions. The program explicitly generates all sum-n compositions over $\{1, 2, 3\}$, and count the ones that satisfy the characterisation. Table 6.3 lists these values for $1 \leq n \leq 30$.

6.3 Augmented Randomised Contraction

Simple Randomised Contraction may produce super-vertices that contain only one vertex, and as discussed in Section 5.2, such a 1-coarse partition-graph is considered an ineffective

Table 6.3: Numbers of possible outcomes of SIMPLE on P_n for $1 \leq n \leq 30$.

n	count	n	count	n	count
1	1	10	46	21	5658
2	1	11	77	22	8615
3	3	12	111	23	13416
4	3	13	181	24	20500
5	6	14	266	25	31829
6	8	15	427	26	48758
7	14	16	636	27	75543
8	19	17	1009	28	115929
9	33	18	1518	29	179344
10	46	19	2388	30	275572

simplification. As a result, in this section we introduce an extension to SIMPLE, called Augmented Contraction (Algorithm 6.7, abbreviated AUG).

During the process of grouping vertices, an unassigned vertex is called *completely free* when all of its neighbours are unassigned, and instead of choosing from all unassigned vertices, AUG chooses only from the vertices that are completely free. A chosen vertex v is grouped with all its neighbours, which are all unassigned because v is completely free.

When there are no more completely free vertices, there may still be vertices that are neither assigned nor completely free. Each of these vertices u has at least one neighbour w that belongs to a super-vertex $\overline{w}$. The second stage of AUG then assigns each u to $\overline{w}$. As a result, the output of AUG is guaranteed to be 1-coarse.

Algorithm 6.7 Augmented Contraction

```
1: procedure AUG(any graph G)
2:     while completely free vertices remain do
3:         Uniformly choose a completely free vertex v
4:         Create a super-vertex of v and all its neighbours      ▷ all neighbours unassigned
5:     end while
6:     for each unassigned vertex u do
7:         u must have a neighbour w that is assigned to a super-vertex w̄
8:         Assign u to w̄
9:     end for
10: end procedure
```

The second stage of AUG is non-deterministic in general. However, if we focus on paths only, then this non-determinism can be removed by modifying AUG into Left-First Augmented Contraction (Algorithm 6.8, abbreviated LEFT-AUG).

Left-First Augmented Contraction also consists of two stages. Its first stage is the same as the first stage of AUG. Then at the end of the first stage, every remaining vertex v_i of the input path must be in either of the following two situations:

1. Its left-hand neighbour v_{i-1} is already assigned (regardless of its right-hand neighbour);

2. Its right-hand neighbour v_{i+1} is already assigned (its left-hand neighbour v_{i-1} is unassigned, or it is v_1 which does not have a left-hand neighbour).

Now, Left-First Augmented Contraction treats the first situation with a higher priority. It always merges a remaining vertex into its left-hand neighbour's super-vertex. A remaining vertex is merged into its right-hand neighbour's super-vertex only if its left-hand neighbour is also unassigned and not completely free, or if it is v_1 and has no left-hand neighbour.

Algorithm 6.8 Left-First Augmented Contraction

```
1: procedure LEFT-AUG(a path P_n)
2:     while completely free vertices remain do
3:         Uniformly choose a completely free vertex v
4:         Create a super-vertex consisting of v and its neighbours
5:     end while
6:     for each unassigned vertex v_i do
7:         if v_i has an assigned left-hand neighbour v_{i-1} then
8:             Merge v_i into the super-vertex containing v_{i-1}
9:         else                          ▷ v_i must have an assigned right-hand neighbour
10:            Merge v_i into the super-vertex containing v_{i+1}.
11:        end if
12:    end for
13: end procedure
```

In a composition that corresponds to an outcome of LEFT-AUG, the possible part-sizes are two, three, four and five. However, not every sum-n composition over $\{2, 3, 4, 5\}$ corresponds to an outcome of LEFT-AUG; for example, LEFT-AUG can never produce 323 on P_8. Similar

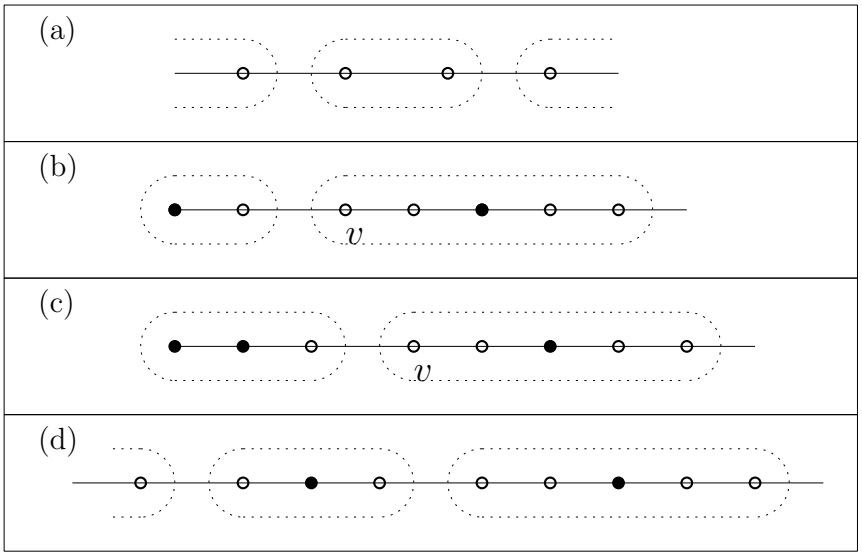


Figure 6.5: Outcomes of LEFT-AUG. (a) 2 cannot occur except at the ends. (b) 25 can only occur at the start. (c,d) 35 can occur only at the start.

to the previous Section 6.2, here we characterise the compositions that represent the possible outcomes of LEFT-AUG.

Definition 6.9. A sum-n composition over $\{2, 3, 4, 5\}$ is called *LA valid* when it satisfies all following three criteria.

(a) 2 can only occur at the endpoints.

(b) 25 does not occur as a substring.

(c) 35 does not occur as substrings except at the left-end.

Theorem 6.10. *A sum-n composition over $\{2, 3, 4, 5\}$ corresponds to an outcome of* LEFT-AUG *if and only if it is LA valid.*

Proof. In the rest of the proof, the term 'composition' means 'composition over $\{2, 3, 4, 5\}$'.

($\Rightarrow$) For a contradiction, assume a composition that is not LA valid. Then there are three possibilities. Firstly, suppose the composition has an occurrence of 2 not at an endpoint, corresponding to Figure 6.5(a). This size-2 super-vertex in the middle has to be created later

than its two neighbouring super-vertices, but neither vertex of this size-2 super-vertex can be chosen because neither is completely free. Hence, 2 cannot occur except at an endpoint.

Secondly, suppose the composition has an occurrence of 25. This 25 must occur at the left-end, because 2 can only occur at an endpoint by the previous paragraph. This situation corresponds to Figure 6.5(b), where the shaded vertices are the only possible choices of starting vertices. However, v has to be merged into its right-hand neighbour, which is not possible under LEFT-AUG.

Thirdly, suppose the composition has an occurrence of 35 not at the left-end. This corresponds to Figure 6.5(d), which is also impossible using similar reasoning above. Meanwhile, note that 35 can occur at the left-end as in Figure 6.5(c).

($\Leftarrow$) To show that every LA valid composition corresponds to an outcome of LEFT-AUG, we use induction on the length of the composition. The base cases are the length-1 compositions 2, 3, 4 and 5. On P_2, P_3 and P_4, LEFT-AUG always groups all vertices into one super-vertex, so the cases of 2, 3 and 4 indeed correspond to outcomes of LEFT-AUG. Meanwhile on P_5, all vertices are grouped into one super-vertex when the first iteration chooses the third vertex of P_5, so the composition 5 also corresponds to an outcome of LEFT-AUG. This completes the base cases.

For the inductive hypothesis, assume that every length-k composition w corresponds to an outcome of LEFT-AUG on P_s, where s denotes the sum of w. Next, consider the composition w' by appending an extra symbol to the end of w, such that w' remains LA valid. As w can end with four possible symbols, the rest of this proof visits each of these four cases.

[Case A] When w ends with 2 and a symbol is to be appended to w, we must assume that w has length one. Otherwise, we would violate the criterion that 2 can only occur at the ends. Hence, there are four subcases: 22, 23, 24 and 25. The first three compositions correspond to choosing v_1 and then v_4 on P_4, P_5 and P_6, respectively. Meanwhile, 25 corresponds to choosing v_1 and then v_5 on P_7.

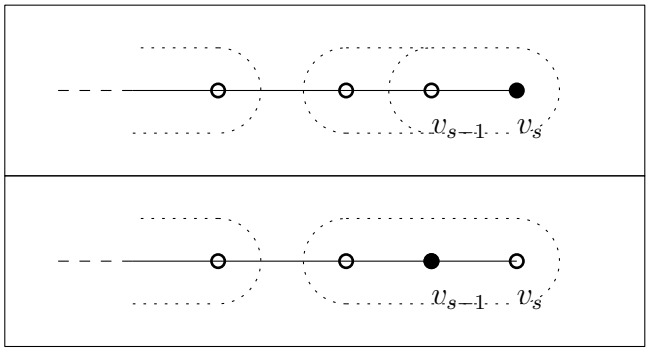


Figure 6.6: Under LEFT-AUG, the starting vertex of the right-most size-3 super-vertex is either v_{s-1} or v_s.

[Case B] When w ends with 3, the starting vertex of the right-most size-3 super-vertex can be either v_{s-1} or v_s, as shown in Figure 6.6. That is, the assumed starting-sequence on P_s contains either v_{s-1} or v_s. If it is v_s that is contained in the starting-sequence, then replacing the occurrence of v_s with v_{s-1} still produces w.

Hence, we can assume that the starting vertex of the right-most size-3 super-vertex is v_s. Then for each of the cases ..32, ..33 and ..34, a corresponding outcome of LEFT-AUG can be obtained by appending v_{s+2} to an existing starting-sequence, as shown in Figure 6.6.

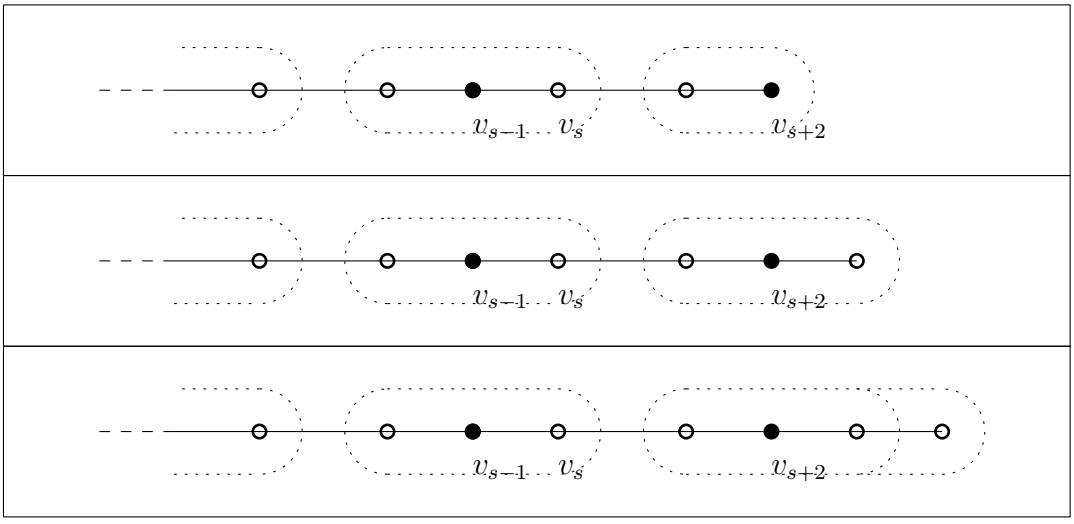


Figure 6.7: The LEFT-AUG extensions when w ends with 3.

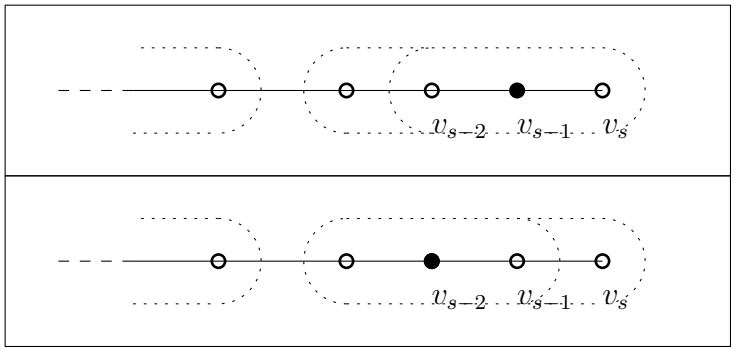


Figure 6.8: Under LEFT-AUG, the starting vertex of the right-most size-4 super-vertex is either v_{s-2} or v_{s-1}.

As for the case that w ends with `3` and we append `5`, we must assume that w has length one, or else we would violate the criterion that `35` can occur only at the start. Then, choosing v_1 and then v_6 is the only possible way that LEFT-AUG produces `35`.

[Case C] When w ends with `4`, the starting vertex of the right-most size-4 super-vertex can be either v_{s-2} or v_{s-1}, as shown in Figure 6.8. If v_{s-1} is the starting vertex, then replacing v_{s-1} with v_{s-2} in the starting-sequence still produces w.

Hence, we can assume that the starting vertex of the right-most size-4 super-vertex is v_{s-2}. Then for each of the compositions `..42`, `..43` and `..44`, a corresponding outcome of LEFT-AUG can be obtained by appending v_{s+2} to an existing starting-sequence. Meanwhile, for the case `..45`, the starting-sequence of a corresponding outcome can be obtained by appending v_{s+3} to the existing starting-sequence.

[Case D] When w ends with `5`, the starting vertex of the right-most super-vertex is necessarily v_{s-2}. Then for each of the cases `..52`, `..53` and `..54`, a corresponding outcome of LEFT-AUG can be obtained by appending v_{s+2} to an existing starting-sequence. Meanwhile, for the case `..45`, a corresponding outcome can be obtained by appending v_{s+3}.

[Conclusion] For the forward direction, we used proof by contradiction to show that every outcome of LEFT-AUG corresponds to an LA valid composition. Then for the backward direction, we used induction on the length of the composition and exhaustive case-analysis to show that every LA valid composition corresponds to an outcome of LEFT-AUG. □

With the characterisation established, we now turn to the enumeration. The notion of LA valid compositions allows us to efficiently compute the number of outcome of LEFT-AUG with a computer program, and Table 6.4 lists the numbers for $1 \leq n \leq 30$. Meanwhile, similar to the end of the previous Section 6.2, we can bound the number of LA valid compositions using simpler sequences.

Every LA valid composition is itself a composition over $\{3, 4, 5\}$ or a composition over $\{2, 3, 4, 5\}$ with a `2` in the first position, the last position, or both. A sum-n composition with the first or last position being a `2` and the remaining positions `3`, `4` or `5` can be viewed as a sum-$(n-2)$ composition over $\{3, 4, 5\}$. Also, a sum-n composition with both endpoints being `2`s and the remaining positions `3`, `4` or `5` can be viewed as a sum-$(n-4)$ composition over $\{3, 4, 5\}$. This leads to an upper bound on the number of sum-n LA valid compositions. Similar to Proposition 6.1, the number u_n of sum-n composition over $\{3, 4, 5\}$ satisfies the following recurrence:

- $u_1 = 0, u_2 = 0, u_3 = 1, u_4 = 1, u_5 = 1$;
- $u_n = u_{n-3} + u_{n-4} + u_{n-5}$ for $5 < n$.

Then the upper bound is $u_n + 2 \cdot u_{n-2} + u_{n-4}$.

As for a lower bound, we note that the following compositions are necessarily LA valid.

(a) Sum-n compositions over $\{4, 5\}$.

(b) Sum-n compositions that start with `24` and the remaining positions `4` or `5`. Their number is the same as the number of sum-$(n-6)$ compositions over $\{4, 5\}$.

(c) Sum-n compositions that end with `2` and the other positions `4` or `5`. Their number is the same as the number of sum-$(n-2)$ compositions over $\{4, 5\}$.

(d) Sum-n compositions that start with `24` and end with `2`, with the other positions being `4` or `5`. Their number is the same as the number of sum-$(n-8)$ compositions over $\{4, 5\}$.

Table 6.4: Numbers of possible outcomes of LEFT-AUG on P_n for $1 \leq n \leq 30$.

n	count	n	count	n	count
1	0	11	12	21	140
2	1	12	15	22	179
3	1	13	19	23	229
4	2	14	25	24	294
5	3	15	32	25	376
6	3	16	40	26	481
7	4	17	52	27	617
8	6	18	67	28	790
9	7	19	85	29	1011
10	9	20	109	30	1295

These four are all subsets of the sum-n LA valid compositions, so the cardinality of their union is a lower bound of the number of sum-n LA valid compositions. The undercounted LA compositions are the ones that include `3`, as the criterion of not containing `35` cannot be concisely captured. Furthermore, we note that these four subsets are disjoint, so the cardinality of the union is simply the sum of the individual cardinalities.

Now, the number ℓ_n of sum-n compositions over $\{4, 5\}$ satisfies the following recurrence:

- $\ell_1 = 0, \ell_2 = 0, \ell_3 = 0, \ell_4 = 1, \ell_5 = 1$;
- $\ell_n = \ell_{n-4} + \ell_{n-5}$ for $5 < n$.

Then the lower bound on the number of sum-n LA valid composition is $\ell_n + \ell_{n-2} + \ell_{n-6} + \ell_{n-8}$.

Finally, note that a simpler pair of bounds consists of sum-n compositions over $\{4, 5\}$ and those over $\{2, 3, 4, 5\}$, but these lead to crude bounds. As a result, we include extra criteria so that the bounds are more accurate.

6.4 Conclusion and Outlook

This chapter is motivated by the fact that a graph can correspond to multiple partition-graphs, and by the desire to understand this set of possible partition-graphs as a whole.

As this goal is challenging on general graphs, we began this line of study with path-graphs, whose partitions can be viewed as integer compositions.

We first studied the set of all 2-sharp 1-coarse partitions on P_n, which correspond to sum-n compositions over $\{2, 3\}$. After stating the basic recurrences for the number and the average length of such compositions, we introduced two notions of the 'balance' of a composition, which is the absolute value of the difference between the sum of the 'left-hand half' and 'right-hand half' of the composition. The plain balance and the gapped balance are the two balance parameters.

The plain balance (Definition 6.4) is of pure combinatorial interest, as it separates the left-hand and right-hand halves of a composition in the intuitive way. Theorem 6.6 then derived a bivariate recurrence $b_p(n, d)$ that counts the number of sum-n compositions over $\{2, 3\}$ with plain balance d.

Meanwhile, the gapped balance (Definition 6.5) is related to the centre-shift. For a composition of even length k, the middle two positions $k/2$ and $k/2 + 1$ are ignored by the gapped balance, so that the resultant value can be used to compute the centre-shift with Lemma 5.22. Theorem 6.7 then derived a bivariate recurrence $b_g(n, d)$ that counts the number of sum-n compositions over $\{2, 3\}$ with gapped balance d.

Many future topics can be pursued from the results in Section 6.1. These include deriving the generating functions and analysing the asymptotic properties of the bivariate sequences $b_p(n, d)$ and $b_g(n, d)$. Bivariate generating functions do not always have closed-form expressions, so this is a challenging goal. Another potential future topic is to extend the notions of balance to other part-sizes. This section only studied compositions with part-sizes two or three, and one can investigate the balance of compositions with other part-sizes.

Section 6.1 did not specify how the partitions of P_n are formed, so Sections 6.2 and 6.3 focused more on the method of construction. In Section 6.2, we introduced the algorithm called SIMPLE (Algorithm 6.6), which is a randomised method that constructs partitions on any graph. We focused on the possible outcomes of SIMPLE on paths, and established

in Theorem 6.8 that the outcomes of SIMPLE applied to P_n are exactly the set of sum-n compositions over $\{1, 2, 3\}$ that contain no substring from $\mathtt{12^*1}$. Next, in Section 6.3, we modified SIMPLE and extended it to LEFT-AUG (Algorithm 6.8), which operates on path-graphs only and constructs 1-coarse partitions. Theorem 6.10 established that the outcomes of LEFT-AUG applied to P_n are exactly the set of sum-n compositions over $\{2, 3, 4, 5\}$ such that (a) 2 can only occur at the endpoints, (b) 25 does not occur as a substring, and (c) 35 does not occur as substrings except at the left-end.

At the end of Sections 6.2 and 6.3, we discussed the enumeration of the two sets of compositions. The enumeration can be further studied, potentially using the Transfer-Matrix Method from the book by Stanley [113, Section 4.7] of the Symbolic Method by Flajolet and Sedgewick [43]. Finally, referring back to this chapter's initial motivation of studying all partition-graphs of a given graph, an ambitious future goal is to use the theory of Random Graphs. Using this deep and powerful theory, one might be able to gain more understanding about the set of all partition-graphs of a general graph.

Chapter 7

Epilogue

Motivated by graph-simplification, the central proposal of this thesis was how quasi-isometries can be used to evaluate the quality of graph-simplification. As the shortest-path distance is a common type of queries on graphs, it is important to measure the amount of distance-distortion between the original graph and the simplified graph. A quasi-isometry bounds the distance-distortion between two graphs up to an additive constant and a multiplicative constant, and thus serves well as a measure of a graph-simplification's effect on the distance-functions. The notion of quasi-isometries also covers the distance-distortion inequalities in many distance-related graph preprocessing methods such as spanners and distance oracles.

Apart from distance-distortion, we were also interested in other graph properties or parameters, and we chose the centre and the median of graphs as the initial properties to study. In order to gather as much information on the centre and the median, Chapter 4 summarised some existing results in a unified notation. Section 4.1 studied the centres of unweighted and edge-weighted trees, and Section 4.2 studied the medians of unweighted, edge-weighted, and vertex-weighted trees. Each of these two sections studied both the structural aspect—characterising the subgraph induced by the centre or the median—and the algorithmic aspects—devising efficient algorithm to locate the centre or the median.

The central theme of this thesis was introduced in Chapter 5. Section 5.1 formally defined quasi-isometries. Then Section 5.2 introduced the simplification method called *partition-graph*, and established some properties related to quasi-isometries, where the quasi-isometry constants depend on the diameters of the partition of the vertices. The rest of Chapter 5 utilised the results surveyed in Chapter 4 to study the centre-related properties of partition-graphs. Section 5.3 defined the new concept of the *centre-shift*, which measures how a general graph-simplification affects the centre. Furthermore, Section 5.3 established an upper bound on the centre-shift, given a quasi-isometry and a special property.

Then, Section 5.4 presented a particular algorithm *Outward-Contraction* to build a partition-graph of a tree. It was shown that *Outward-Contraction* always produces a tree with centre-shift zero. In other words, *Outward-Contraction* is a simplification method that preserves the centres of trees. Section 5.5 turned to the medians of trees. Provided that we store the cardinalities of the super-vertices of partition-graph of a tree, and that we treat the partition-graph as having vertex-weights, then Section 5.5 showed that any partition-graph preserves the medians of trees.

A graph has multiple possible partition-graphs, so we were interested in studying such a set of possible outcomes of a fixed graph. To simplify matters, we started with path-graphs. Chapter 6 studied the set of partition-graphs of P_n under three different settings. Under the first setting, studied in Section 6.1, every super-vertex is assumed to have either two or three vertices. From this we derived a recurrence to count the total number of *balanced* partitions, which in turn is related to the centre-shift of partition-graphs of paths. The second and third settings are respectively studied in Sections 6.2 and 6.3. Each of these two sections presented a probabilistic algorithm for generating partitions on paths, and characterised the possible outcomes in terms of a regular language.

At the end, despite starting with just one general idea of graph-simplification with quasi-isometries, the past three years or so had seen this thesis growing into the three branches of Chapters 4, 5 and 6, each with its own future directions. Now, having taken a final glimpse of these pages and noted the future possibilities, perhaps here is a suitable place to close. As for any criticism or new idea, the author now leaves to the esteemed readers.

Sans plus il faut dormir en l'oubli du blasphéme,
Sur le sable altéré gisant et comme j'aime
Ouvrir ma bouche à l'astre efficace des vins!

Couple, adieu; je vais voir l'ombre que tu devins.

—Mallarmé (The Afternoon of a Faun) [1]

[1] And so I must sleep, in oblivion of blasphemy, / Lying on the thirsty sand and as I please / To open my mouth to the fruitful star of wine! / Couple, farewell; I'll see the shadow you became.

Bibliography

[1] I. Abraham, Y. Bartal, and O. Neiman. Advances in metric embedding theory. *Advances in Mathematics*, 228(6):3026–3126, 2011.

[2] R. Ahmed, G. Bodwin, F. D. Sahneh, K. Hamm, M. J. L. Jebelli, S. Kobourov, and R. Spence. Graph spanners: A tutorial review. *Computer Science Review*, 37, 2020.

[3] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st Symposium on Principles of Database Systems (PODS)*, pages 5–14, 2012.

[4] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28(4):1167–1181, 1999.

[5] Z. Allen-Zhu, Z. Liao, and L. Orecchia. Spectral sparsification and regret minimization beyond matrix multiplicative updates. In *STOC*, pages 237–245, 2015.

[6] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete and Computational Geometry*, 9(1):81–100, 1993.

[7] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39, 2008.

[8] S. Baswana and T. Kavitha. Faster algorithms for all-pairs approximate shortest paths in undirected graphs. *SIAM Journal on Computing*, 39(7):2865–2896, 2006.

[9] S. Baswana, T. Kavitha, K. Mehlhorn, and S. Pettie. Additive spanners and (α, β)-spanners. *ACM Transactions on Algorithms*, 7(1):5:1–5:26, 2010.

[10] S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in expected $o(n^2)$ time. *ACM Transactions on Algorithms*, 2(4):557–577, 2006.

[11] J. D. Batson, D. A. Spielman, and N. Srivastava. Twice-Ramanujan sparsifiers. *SIAM Journal on Computing*, 41(6):1704–1721, 2012.

[12] A. A. Benczúr and D. R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In *STOC*, pages 47–55, 1996.

[13] A. A. Benczúr and D. R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *SIAM Journal on Computing*, 44(2):290–319, 2015.

[14] P. Berman, A. Bhattacharyya, K. Makarychev, S. Raskhodnikova, and G. Yaroslavtsev. Approximation algorithms for spanner problems and Directed Steiner Forest. *Information and Computation*, 222:93–107, 2013.

[15] A. Bernstein, K. Däubel, Y. Disser, M. Klimm, T. Mütze, and F. Smolny. Distance-preserving graph contractions. *SIAM Journal on Discrete Mathematics*, 33(3):1607–1636, 2019.

[16] M. Besta and T. Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations. arXiv:1806.01799, 2018.

[17] M. Besta, S. Weber, L. Gianinazzi, R. Gernstenberger, A. Ivanov, Y. Oltchik, and T. Hoefler. Slim Graph: Practical lossy graph compression for approximate graph processing, storage, and analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–25, 2019.

[18] C. Bichot and P. Siarry, editors. *Graph Partitioning.* Wiley, 2013.

[19] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[20] A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph Classes: A Survey*. SIAM, 1999.

[21] F. Buckley, Z. Miller, and P. J. Slater. On graphs containing a given graph as center. *Journal of Graph Theory*, 5(4):427–434, 1981.

[22] T. M. Chan and D. Skrepetos. Faster approximate diameter and distance oracles in planar graphs. *Algorithmica*, 81(8):3075–3098, 2019.

[23] G. J. Chang. Centers of chordal graphs. *Graphs and Combinatorics*, 7:305–313, 1991.

[24] P. Charalampopoulos, P. Gawrychowski, S. Mozes, and O. Weimann. Almost optimal distance oracles for planar graphs. In *STOC*, pages 138–151, 2019.

[25] S. Chechik. New additive spanners. In *ACM-SIAM on Discrete Algorithms*, pages 498–512, 2013.

[26] S. Chechik. Approximate distance oracles with improved bounds. In *STOC*, pages 1–10, 2015.

[27] V. Chepoi. Centers of triangulated graphs. *Mathematical Notes of the Academy of Sciences of the USSR*, 43(1):82–86, 1988.

[28] V. Chepoi and F. Dragan. A linear-time algorithm for finding a central vertex of a chordal graph. In *European Symposium on Algorithms*, pages 159–170, 1994.

[29] E. Chlamtáč, M. Dinitz, G. Kortsarz, and B. Laekhanukit. Approximating spanners and Directed Steiner Forest: Upper and lower bounds. *ACM Transactions on Algorithms*, 16(3), 2020. Article 33.

[30] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009. 3rd ed.

[32] D. Cvetković, P. Rowlinson, and S. Simić. *An Introduction to the Theory of Graph Spectra*, volume 75 of *London Mathematical Society Student Texts*. Cambridge University Press, 2010.

[33] M. Cygan, F. Grandoni, and T. Kavitha. On pairwise spanners. In *STACS*, 2013.

[34] M. S. Daskin and K. L. Maass. The p-median problem. In G. Laporte, S. Nickel, and F. S. da Gama, editors, *Location Science*, chapter 2, pages 21–45. Springer, 2015.

[35] R. Diestel. *Graph Theory*. Springer, 5th edition, 2017.

[36] A. Dvoretzky and H. Robbins. On the "parking" problem. *Publications of the Mathematical Research Institute of the Hungarian Academy of Sciences*, 9:109–225, 1964.

[37] M. Elkin and D. Peleg. $(1 + \epsilon, \beta)$-spanner construction for general graphs. *SIAM Journal on Computing*, 33(3):608–631, 2004.

[38] M. Elkin and D. Peleg. The hardness of approximating spanner problems. *Theory of Computing Systems*, 41:691–729, 2007.

[39] A. M. Farley. Broadcast time in communication networks. *SIAM Journal on Applied Mathematics*, 39(2):385–390, 1980.

[40] A. M. Farley and A. Proskurowski. Computation of the center and diameter of outerplanar graphs. *Discrete Applied Mathematics*, 2(3):185–191, 1980.

[41] K. Faust and S. Wassermann. *Social network analysis: methods and applications*. Cambridge University Press, 1994.

[42] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23(98):298–305, 1973.

[43] P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. Cambridge, 2009.

[44] S. Fortunato. Community detection in graphs. *Physics Reports*, 486:75–174, 2010.

[45] S. Fortunato and D. Hric. Community detection in networks: A user guide. *Physics Reports*, 659:1–44, 2016.

[46] K. Georgiou, E. Kranakis, and D. Krizanc. Random maximal independent sets and the unfriendly theater seating arrangement problem. *Discrete Mathematics*, 309(16):5120–5129, 2009.

[47] A. J. Goldman. Optimal locations for centers in networks. *Transportation Science*, 3(4):352–360, 1969.

[48] A. J. Goldman. Optimal center location in simple networks. *Transportation Science*, 5(2):212–221, 1971.

[49] A. J. Goldman. Minimax location of a facility in a network. *Transportation Science*, 6(4):407–148, 1972.

[50] N. Graham, R. C. Entringer, and L. A. Székely. New tricks for old trees: Maps and the pigeonhole principle. *The American Mathematical Monthly*, 101(7):664–667, 1994.

[51] D. Granot and D. Skorin-Kapov. On some optimization problems on k-trees and partial k-trees. *Discrete Applied Mathematics*, 48(2):129–145, 1994.

[52] M. Gromov. Groups of polynomial growth and expanding maps. *Publications Mathématiques de l'IHÉS*, 53:53–78, 1981.

[53] Q.-P. Gu and G. Xu. Constant query time $(1 + \epsilon)$-approximate distance oracle for planar graphs. *Theoretical Computer Science*, 761:78–88, 2019.

[54] S. L. Hakimi. Optimum locations of switching centres and the absolute centers and medians of a graph. *Operations Research*, 12(3):450–459, 1964.

[55] S. Halfin. Letter to the editor—on finding the absolute and vertex centers of a tree with distances. *Transportation Science*, 8(1):75–77, 1974.

[56] G. Y. Handler. Minimax location of a facility in an undirected tree graph. *Transportation Science*, 7(3):287–293, 1973.

[57] F. Harary. Status and contrastatus. *Sociometry*, 22(1):23–43, March 1959.

[58] F. Harary and P. A. Ostrand. The cutting center theorem for trees. *Discrete Mathematics*, 1:7–18, 1971.

[59] F. Harary and P. J. Slater. A linear algorithm for the cutting center of a tree. *Information Processing Letters*, 23:317–319, 1986.

[60] S. M. Hedetniemi, E. J. Cockayne, and S. T. Hedetniemi. Linear algorithms for finding the Jordan center and path center of a tree. *Transportation Science*, 15(2):98–114, 1981.

[61] S. Heubach and T. Mansour. *Combinatorics of Compositions and Words*. CRC Press, 2010.

[62] J. L. Jackson and E. W. Montroll. Free radical statistics. *The Journal of Chemical Physics*, 28(6):1101–1109, 1958.

[63] C. Jordan. Sur les assemblages des lignes. *Journal für die reine und angewandte Mathematik*, 1869:185–190, 1869.

[64] R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *Journal of the ACM*, 51(3):497–515, 2004.

[65] O. Kariv and S. L. Hakimi. An algorithmic approach to network location problems. I: The p-centers. *SIAM Journal on Applied Mathematics*, 37(3):513–538, 1979.

[66] T. Kavitha. New pairwise spanners. *Theory of Computing Systems*, 61(4):1011–1036, 2017.

[67] T. Kavitha and N. M. Varma. Small stretch pairwise spanners. In *Automata, Languages, and Programming (ICALP). Lecture Notes in Computer Science*, volume 7965, pages 601–612. Springer, 2013.

[68] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, 1970.

[69] B. Khoussainov and T. Takisaka. Large scale geometries of infinite strings. In *LICS*, 2017.

[70] J. Konegis. KONECT: the Koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350, 2013.

[71] G. Kortsarz. On the hardness of approximating spanners. In *APPROX*, pages 135–146, 1998.

[72] M. Krivelevich, T. Mészáros, P. Michaeli, and C. Shikhelman. Greedy maximal independent sets via local limits. In *The 31st International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms (AofA, LIPICS)*, volume 159, pages 20:1–20:19, 2020. Leibniz International Proceedings in Informatics.

[73] B. Krön and R. G. Möller. Quasi-isometries between graphs and trees. *Journal of Combinatorial Theory, Series B*, 98(5):994–1013, 2008.

[74] R. Laskar and D. Shier. On powers and centers of chordal graphs. *Discrete Applied Mathematics*, 6(2):139–147, 1983.

[75] H.-Y. Lee and G. J. Chang. The w-median of a connected strongly chordal graph. *Journal of Graph Theory*, 18(7):673–680, 1994.

[76] Y. T. Lee and H. Sun. Constructing linear-sized spectral sparsification in almost-linear time. *SIAM Journal on Computing*, 47(6):2315–2336, 2018.

[77] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15(2):215–245, 1995.

[78] L. Lovász. *Combinatorial Problems and Exercises*. Elsevier, second edition, 1993.

[79] J. K. Mackenzie. Sequentially filling of a line by intervals placed at random and its application to linear absorption. *Journal of Chemical Physics*, 37(4):723–728, 1962.

[80] S. Mandal, A. Pal, and M. Pal. An optimal algorithm to find centres and diameter of a circular-arc graph. *Advanced Modeling and Optimization*, 9(1):155–170, 2007.

[81] A. McGregor. Graph stream algorithms: A survey. *SIGMOD Record*, 43(1):9–20, 2014.

[82] N. Megiddo. Linear-time algorithms for linear programming in $\mathbb{R}^3$ and related problems. *SIAM Journal of Computing*, 12(4):759–776, 1983.

[83] F. Menczer, S. Fortunato, and C. A. Davis. *A First Course in Network Science*. Cambridge University Press, 2020.

[84] M. Mendel and A. Naor. Ramsey partitions and proximity data structures. *Journal of the European Mathematical Society*, 9(2):253–275, 2007.

[85] M. Miller and Širán. Moore graphs and beyond: A survey of the degree/diameter problem. *The Electronic Journal of Combinatorics*, pages DS14–May, 2013.

[86] E. Minieka. A polynomial time algorithm for finding the absolute center of a network. *Networks*, 11(4):351–355, 1981.

[87] N. Mishra, R. Schreiber, I. Stanton, and R. E. Tarjan. Clustering social networks. In *The 5th International Workshop on Algorithms and Models for the Web-Graph*, pages 56–67, 2007.

[88] N. Mishra, R. Schreiber, I. Stanton, and R. E. Tarjan. Finding strongly knit clusters in social networks. *Internet Mathematics*, 5(1-2):155–174, 2011.

[89] N. Mladenović, J. Brimberg, P. Hansen, and J. A. Moreno-Pérez. The p-median problem: A survey of metaheuristic approaches. *European Journal of Operational Research*, 179(3):927–939, 2007.

[90] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD*, pages 419–432, 2008.

[91] S. Olariu. A simple linear-time algorithm for computing the center of an interval graph. *International Journal of Computer Mathematics*, 34:121–128, 1990.

[92] M. I. Ostrovskii. *Metric Embeddings*. De Gruyter, 2013.

[93] E. S. Page. The distribution of vacancies on a line. *Journal of the Royal Statistical Society: Series B*, 21(2):364–374, 1959.

[94] D. Peleg and A. A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[95] A. Proskurowski. Centers of maximal outerplanar graphs. *Journal of Graph Theory*, 4(1):75–79, 1980.

[96] M. Pătraşcu and L. Roditty. Distance oracles beyond the Thorup-Zwick bound. *SIAM Journal on Computing*, 43(1):300–311, 2014.

[97] K. B. Reid. Centroids to centers in trees. *Networks*, 21:11–17, 1991.

[98] K. B. Reid. Centrality measures in trees. In H. Kaul and H. M. Mulder, editors, *Advances in Interdisciplinary Applied Discrete Mathematics*, chapter 8, pages 167–197. World Scientific, 2011.

[99] A. Rényi. On a one-dimensional problem concerning random space-filling problem. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 3:109–127, 1958.

[100] G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.

[101] A. E. Sarıyüce, K. Kaya, E. Saule, and Ü. V. Çatalyürek. Graph manipulations for fast centrality computation. *ACM Transactions on Knowledge Discovery from Data*, 11(3):26:1–26:25, 2017.

[102] S. E. Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.

[103] K. Shin, A. Ghoting, M. Kim, and H. Raghavan. SWeG: Lossless and lossy summarization of web-scale graphs. In *The World Wide Web Conference*, pages 1679–1690, 2019.

[104] P. J. Slater. Centers to centroids in graphs. *Journal of Graph Theory*, 2:209–222, 1978.

[105] P. J. Slater. Medians of arbitrary graphs. *Journal of Graph Theory*, 4(4):389–392, 1980.

[106] P. J. Slater. The k-nucleus of a graph. *Networks*, 11:233–242, 1981.

[107] P. J. Slater. Locating central paths in a graph. *Transportation Science*, 16:1–18, 1982.

[108] C. Smart and P. J. Slater. Center, median, and centroid subgraphs. *Networks*, 34(4):303–311, 1999.

[109] H. Solomon and H. Weiner. A review of the packing problem. Technical report, Department of Statistics, Stanford University, 1986.

[110] C. Sommer, E. Verbin, and W. Yu. Distance oracles for sparse graphs. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 703–712, 2009.

[111] D. A. Spielman and N. Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011.

[112] D. A. Spielman and S.-H. Teng. Spectral sparsification of graphs. *SIAM Journal on Computing*, 40(4):981–1025, 2011.

[113] R. P. Stanley. *Enumerative Combinatorics*, volume 1. Cambridge University Press, second edition, 2011.

[114] B. Tansel. Discrete center problems. In H. A. Eiselt and V. Marianov, editors, *Foundations of Location Analysis*, chapter 5, pages 79–106. Springer, 2011.

[115] P. Tatikonda, H. Kuppili, A. Paila, J. Pail, and J. J. Nair. Construction of multiplicative graph spanners using minimum connected dominating set with bounded diameter. In *International Conference on Advances in Computing, Communicationsand Informatics (ICACCI)*, 2018.

[116] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM*, 51(6):993–1024, 2004.

[117] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.

[118] H. Toivonen, F. Zhou, A. Hartikainen, and A. Hinkka. Compression of weighted graphs. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 965–973, 2011.

[119] Y. Veluri, P. Jayakumar, and J. J. Nair. Computing multiplicative spanners efficiently for a class of simple graphs. In *International Conference on Data Science and Engineering (ICDSE)*, 2018.

[120] J. J. Whang, D. F. Gleich, and I. S. Dhillon. Overlapping community detection using seed set expansion. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*, pages 2099–2108, 2013.

[121] H. Wittenberg. Local medians in chordal graphs. *Discrete Applied Mathematics*, 28(3):287–296, 1990.

[122] C. Wulff-Nilsen. Approximate distance oracles with improved preprocessing time. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 202–208, 2012.

[123] C. Wulff-Nilsen. Approximate distance oracles with improved query time. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 539–549, 2013.

[124] H.-G. Yeh and G. J. Chang. Centers and medians of distance-hereditary graphs. *Discrete Mathematics*, 265:297–310, 2003.

[125] B. Zelinka. Medians and peripherians of trees. *Archivum Mathematicum*, 4(2):87–95, 1968.