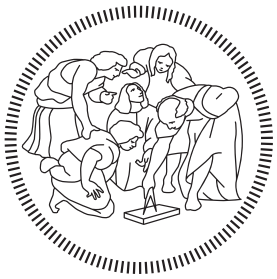
POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Master of Science in
Computer Science and Engineering



Advances in Graph Partitioning Algorithms

Advisor: PROF. MATTEO MATTEUCCI
Co-advisor: ANDREA ROMANONI

Master Graduation Thesis by:

MATTEO FROSI
Student Id n. 875393

Academic Year 2017-2018

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

This template has been adapted by Emanuele Mason, Andrea Cominola and Daniela Anghileri: *A template for master thesis at DEIB*, June 2015.

ACKNOWLEDGMENTS

This work has been possible thanks to my advisor, Matteo, that provided me the laptop used during the whole thesis, and my co-advisor, Andrea. They both introduced me to the big world of scientific researches, quite different from the school one lived until the last year, and helped me in developing ideas and skills.

I want to thanks all the people that supported me during this intense year and the preceding ones. I am grateful to my mother Sandrina, who helped me during my whole life, and it is she who raised and allowed me to be the person I am now. I thank my aunts, Anna and Angela, that week after week made me laugh and spoiled me in every possible way. I thank my best friend Valentina, with whom I spoke, taught, laughed and fought, for every single day of the last three years. I also thank my friend Luca that, for the two years spent going back and forth to Milano by train, supported me, living the same university experience. Lastly, but not least important, I want to remember and thank Zeus the old cat, that filled me and my family with joy and love for the last six months, before passing away.

ABSTRACT

Graph partitioning is a problem that interested scientists since the last century. Many real-world applications can be modelled as graphs: social networks, telecommunication routing, images, circuitry design, etc. Over the course of the last 60 years, an increasing variety of algorithm has been proposed to solve the graph partitioning problem, adapting to its evolution. If at the beginning the main goal was to divide graphs into equal sized groups, the explosion of data occurred in the last decade shifted the focus on scalable methods, capable of working on large graphs, trying to maintain balanced solutions.

In this thesis, three new graph partitioning algorithms are presented, having different purposes and working principles. These algorithms were thought with multiple goals. First, they complete unused ideas, never fully explored in the literature or that were simply discarded due to their inefficiency. Second, these algorithms are created to reach results comparable with state of the art algorithms, in terms of speed and quality. The last, and main, purpose is to find solutions to specific real-world problems, as mesh partitioning or image segmentation, that in the last years moved towards real-time requirements. With these algorithms, we were able to achieve three results: a fast and scalable hybrid multi-level clustering; an incredibly fast tree based partitioning, performing better than the state of the art algorithms, and a slightly slower method, derived from the depth first search on graphs, to obtain perfectly balanced partitions.

Our partitioning algorithms are tested against meshes derived from the images belonging to publicly available EPFL dataset [3] and the images used by the 3D reconstruction suite COLMAP [1, 2]; the mesh model collection provided with the benchmark described in [4]; randomly generated labelled graphs and images and some of the segmented images of the KITTI Vision benchmark suite [5, 82].

SOMMARIO

Un gran numero di situazioni del mondo reale può essere descritto attraverso grafi. Un grafo è un insieme di nodi interconnessi da lati. I nodi sono di solito una rappresentazione astratta di oggetti e i lati sono le loro relazioni. Uno dei problemi più studiati e documentati relativo ai grafi è il loro partizionamento.

Il partizionamento di grafi viene utilizzato in molte applicazioni: i social network fanno affidamento su di esso per elaborare richieste da parte di utenti su sottoinsiemi di grafi più piccoli, distribuendo il carico di lavoro su più dispositivi; le reti biologiche, che possono essere rappresentate utilizzando l'astrazione del grafo (ad esempio l'interazione proteina-proteina o i processi metabolici), usano il partizionamento per risolvere i problemi di interazione biologica, rilevando comunità e gruppi di elementi simili; la pianificazione del percorso nelle reti di trasporto sfrutta il partizionamento per ottenere una maggiore velocità, utilizzando approcci di divisione e conquista (si pianifica un percorso su regioni più piccole e poi i risultati sono combinati); i sistemi VLSI (integrazione su larga scala) fanno uso del partizionamento di grafi per ridurre la connessione tra i circuiti nella loro progettazione; in computer vision le immagini sono rappresentate da grafi in cui i pixel sono nodi, e sono partizionati in base alla loro somiglianza; nella ricostruzione 3D, le superfici sono modellate come grafi, che devono essere partizionati per applicare procedure di raffinamento in tempi ragionevoli.

La grande quantità di applicazioni è la ragione principale per cui, dal secolo scorso, il problema del partizionamento di grafi coinvolse un numero crescente di ricercatori ed esperti. Nel corso degli ultimi 60 anni, è stata proposta una notevole varietà di algoritmi. Molti sono gli scopi di questi metodi, come la scalabilità, il bilanciamento, la minimizzazione del taglio dei bordi e una veloce computazione. Diversi approcci si sono evoluti, cercando di risolvere le molteplici sfaccettature del problema di partizionamento di grafi.

I primi grafi considerati furono quelli i cui nodi potevano essere rappresentati con coordinate geometriche. Per suddividere questo tipo di strutture, sono stati sviluppati algoritmi basati sulla geometria, il cui principio di funzionamento è quello di dividere ricorsivamente il dominio dei nodi in più parti. Sebbene questi algoritmi producano partizioni perfettamente bilanciate, avendo tutti la stessa quantità di nodi, sono afflitti da molteplici inconvenienti. Innanzitutto, non considerano la struttura del grafo, basandosi solo sulle coordinate geometriche dei nodi: questo porta al partizioni contenente componenti disconnessi. La Figura 0.1 mostra un esempio di questo prob-

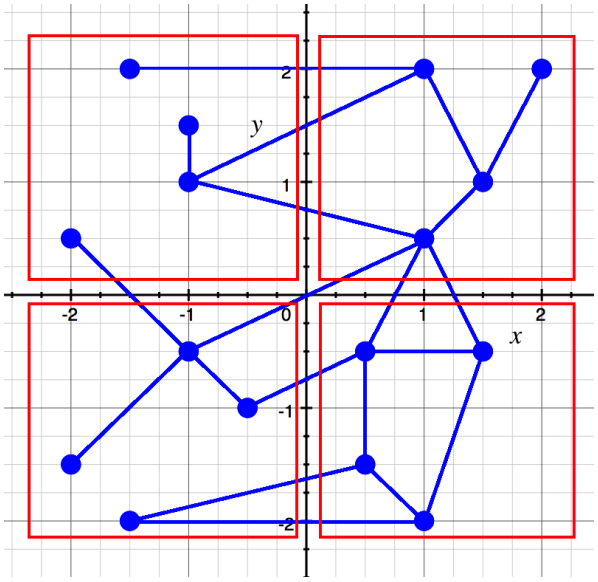


Figure 0.1: Uno dei problemi degli algoritmi di partizionamento geometrici è che le partizioni ottenute sono formate da nodi disconnessi.

lema: le partizioni a sinistra, rappresentate come quadrati rossi, contengono sottoinsiemi di nodi scollegati. In secondo luogo, con l'aumentare del numero di coordinate, i metodi geometrici diventano più lenti. Infine, essi sono confinati a grafi i cui nodi possono essere rappresentati nello spazio Euclideo.

Successivamente, i lavori si spostarono verso i metodi spettrali, che invece sfruttano l'insieme di lati nel grafo, ma possono essere applicati solo su piccole strutture a causa della loro complessità temporale. Gli algoritmi di partizionamento spettrale si basano su una delle rappresentazioni matematiche di un grafo, la matrice di adiacenza, per studiarne le proprietà di connettività e per eseguire partizioni basandosi su di esse. Sebbene i metodi spettrali si traducano in partizioni ottimali, sia in termini di dimensioni che di numero di lati che le attraversano, non sono scalabili, diventando impossibili da adottare anche su grafi di medie dimensioni (migliaia di nodi).

Nell'ultimo decennio, la maggior parte degli algoritmi si basò su euristiche a più livelli, utilizzate per partizionare grafi di dimensioni considerevoli. L'idea alla base di questi approcci è semplice e consiste in tre passaggi consecutivi. In primo luogo, le dimensioni di un grafo sono ridotte, aggregando nodi che soddisfano un criterio specifico: peso massimo del lato che li collega, un numero simile di nodi adiacenti, ecc. Questo passo viene ripetuto, approssimando il grafo iniziale fino a quando un nuovo grafo con dimensioni sufficientemente piccole è ottenuto. Quindi, un algoritmo di partizionamento viene applicato sul piccolo grafo, producendo risultati accurati in breve tempo. Infine, le partizioni sono raffinate, propagando

l'assegnazione di ciascun nodo dal grafio piccolo a quello iniziale. Questi algoritmi producono risultati di alta qualità in tempi molto brevi, con partizioni quasi perfettamente bilanciate, che superano i metodi geometrici e spettrali. Devono, tuttavia, essere regolati con precisione in ogni fase, considerando anche le possibili correlazioni tra esse. Ad esempio, se la seconda fase viene eseguita con un algoritmo che sacrifica la qualità del partizionamento per ottenere un tempo di esecuzione ridotto, il raffinamento dovrebbe essere forte e preciso e viceversa. Al giorno d'oggi, l'obiettivo principale è quello di sviluppare algoritmi sempre più scalabili e veloci, adottando tecniche di parallelizzazione, per muoversi verso le applicazioni in tempo reale.

CONTRIBUTO DELLA TESI

Proponiamo tre nuovi algoritmi di partizionamento grafi con diversi principi di funzionamento e associati a situazioni distinte. Il primo algoritmo, Raggruppamento di Etichette (Label Clustering), ha l'obiettivo di ottenere un partizionamento di grafo basato sulla somiglianza dei nodi, seguendo un approccio multilivello. Un grafo viene prima raggruppato tramite aggregazione dei nodi aventi la massima somiglianza, utilizzando una versione modificata della Visita in Profondità su grafi, quindi viene ridotto di dimensioni fino a ottenere una struttura sufficientemente piccola per applicare un metodo di partizionamento diretto.

Il secondo algoritmo, il Partizionamento tramite Albero AD (AD-tree partitioning), sfrutta una nuova struttura dati per eseguire un partizionamento rapido (lineare nel numero di nodi). La struttura menzionata è l'unione di tre parti: un grafo diretto con più componenti connessi (chiamato grafo dei discendenti), un albero normale derivato da un grafo e un albero diretto (chiamato albero di Arianna), i cui nodi sono collegati come se fossero parte di una catena (nel senso che ogni nodo, tranne il primo e l'ultimo, ha un lato che punta a un solo altro nodo ed è puntato solo da un lato). L'algoritmo è basato sulla propagazione del valore dei nodi dell'albero AD e sul loro attraversamento, consentendo di tagliare gruppi di nodi con una velocità superiore agli algoritmi allo stato dell'arte, indipendentemente dal numero di partizioni desiderate.

L'ultimo algoritmo, chiamato Partizionamento Orientato (Directed partitioning), non si basa su alcuna struttura o similarità dei dati, ma funziona direttamente sul grafo durante la sua esplorazione. L'idea alla base è quella di utilizzare una procedura di visita simile alla Visita in Profondità, per ottenere partizioni perfettamente bilanciate, avendo anche un'alta connettività interna. Diversi problemi derivano dalla Visita in Profondità, come la formazione di buchi nel grafo, causati principalmente dalla sua complessità strutturale. Partizionamento Orientato risolve questi problemi, al costo di una prestazione temporale leggermente peggiore rispetto all'algoritmo di

partizionamento basato sull'Albero AD, ma fornendo partizioni perfettamente bilanciate.

Riassumendo, i tre algoritmi proposti contribuiscono in modi differenti alla letteratura relativa al partizionamento di grafi. Il primo partiziona grafi con un approccio ibrido multi-livello, ottenendo gruppi contenenti nodi con alta somiglianza, in tempi ragionevoli (pochi secondi per raggruppare grafi con milioni di nodi e più lati). Il secondo sfrutta una struttura dati avanzata per ottenere partizioni in un tempo di un ordine di grandezza inferiore al tempo impiegato dagli algoritmi allo stato dell'arte, sacrificando un certo bilanciamento nelle partizioni. Il terzo ottiene, in tempi brevi, blocchi perfettamente bilanciati, una delle caratteristiche più desiderate di un algoritmo di partizionamento grafi.

DESCRIZIONE DELLA TESI

La tesi è organizzata in due parti. Nella prima, forniamo i materiali di base necessari per comprendere i contributi della tesi.

- Il Capitolo 2 introduce alcuni concetti matematici, come definizioni e teoremi relativi a grafi e partizionamento di grafi. Poi viene presentata una revisione della letteratura, descrivendo l'evoluzione degli algoritmi utilizzati per il partizionamento di grafi, concentrandosi su alcuni di essi.

La seconda parte contiene il contributo principale di questa tesi, descrivendo tre nuovi metodi per il partizionamento di grafi.

- Il Capitolo 3 illustra l'algoritmo multi-livello Label Clustering. Dopo aver definito nuovi concetti matematici, ci concentriamo sulla descrizione di ciascuna fase dell'algoritmo. Innanzitutto, consideriamo la riduzione iniziale di un grafo, il cui scopo è quello di costruire, da esso, un nuovo grafo tale che due suoi nodi adiacenti con la stessa etichetta non esistano. Viene proposto poi un semplice schema di approssimazione, per ridurre ulteriormente la dimensione del grafo ridotto. Infine, descriviamo il metodo utilizzato per partizionare il più piccolo grafo ottenuto.

- Il Capitolo 4 presenta una nuova struttura dati per rappresentare grafi, l'albero AD e un algoritmo di partizionamento basato sul suo attraversamento. L'albero AD deriva da altre tre strutture, ciascuna descritta in dettaglio: l'albero corrispondente al grafo; il grafo formato da tutti i nodi e i bordi non inseriti nell'albero, detto grafo dei discendenti; l'albero che rappresenta l'ordine di inserimento dei nodi del grafo nella prima struttura, detto albero di Arianna. Dopo una panoramica delle proprietà dell'albero AD e dei metodi di creazione, l'attenzione

si sposta sull'algoritmo di partizionamento, presentando e descrivendo ogni routine che lo compone.

- Il Capitolo 5 rivisita la procedura di Visita in Profondità su grafi, derivando un nuovo algoritmo, denominato Partizionamento Orientato, che sfrutta la lista di adiacenza dei nodi del grafo per ottenere partizioni perfettamente bilanciate aventi anche forme regolari. In particolare, mostriamo alcuni dei problemi legati all'uso della Visita in Profondità, usata per il partizionamento, risolvendoli e adattandoli all'algoritmo proposto.

- Il Capitolo 6 è diviso in due sezioni. Nella prima si considera un caso d'uso specifico in cui gli algoritmi di partizionamento di grafi possono essere utilizzati per migliorare i risultati dello stato dell'arte: la ricostruzione 3D. La sezione introduce brevemente il problema della ricostruzione e la letteratura che ne è associata, concentrandosi su algoritmi recenti. Poi, mostriamo come i nostri metodi di partizionamento di grafi possono essere utilizzati per ridurre il tempo necessario per ottenere ricostruzioni accurate.

  La seconda sezione include tre miglioramenti dell'algoritmo di partizionamento dell'albero AD, descritto nel Capitolo 4, tutti basati sul grafo dei discendenti. Il primo è una tecnica di pre-elaborazione, che migliora il bilanciamento della struttura dati ottenuta dal grafo (l'abero AD). Il secondo è un'euristica greedy usata per supportare l'algoritmo di partizionamento, aumentandone le possibilità di taglio. L'ultimo miglioramento, di post-elaborazione, si basa sullo scambio di elementi tra le partizioni, ancora una volta per migliorarne il bilanciamento complessivo.

Infine, la tesi termina rivisitando brevemente il problema del partizionamento di grafi e i tre algoritmi, compresi possibili miglioramenti e lavori futuri.

CONTENTS

1 INTRODUCTION

A large number of real-world situations can be described with graphs. A graph is a set of nodes interconnected by edges. Nodes are usually an abstract representation for objects, and edges for their relations. One of the most studied and documented problem related to graphs is their partitioning.

Graph partitioning is used in many applications: social networks rely on it in order to process user queries on smaller subsets of graphs, distributing the workload on multiple devices; biological networks, that can be represented by using the graph abstraction (for example protein-protein interaction or metabolic processes), use partitioning to solve biological interaction problems, detecting communities and groups of similar elements; route planning in transportation networks exploits partitioning to obtain a speed up, using divide and conquer approaches (plan a route on smaller regions and then combine the results); VLSI (very large-scale integration) systems make use of graph partitioning to reduce the connection between circuits in designing them; in computer vision images are represented as graphs where the pixel are nodes, and they are segmented according to their similarity; in 3D reconstruction, surface meshes are modelled as graphs, which have to be partitioned to apply refinement procedures in reasonable times.

The great amount of applications is the main reason that, since the last century, the graph partitioning problem involved an increasing number of researchers and experts. Over the course of the last 60 years, a considerable variety of algorithms has been proposed. Many are the purposes of these methods, such as scalability, balancing, minimisation of the edge cut and fast computation. Different approaches evolved, trying to solve the multiple facets of the graph partitioning problem.

The first graphs to be considered were the ones whose nodes could be represented with coordinates. To partition these kind of structures, geometry based algorithm were developed, whose working principle is to recursively split the node domain in multiple parts. Although these algorithms produce perfectly balanced partitions, having all the same amount of nodes, they are afflicted by multiple drawbacks. First, they do not consider the graph structure, relying only on the geometrical coordinates of nodes: this leads to partitions containing disconnected components. Figure 1.1 shows an example of this problem: the partitions on the left, represented as red squares, contain subsets of disconnected nodes. Second, as the number of coordinates

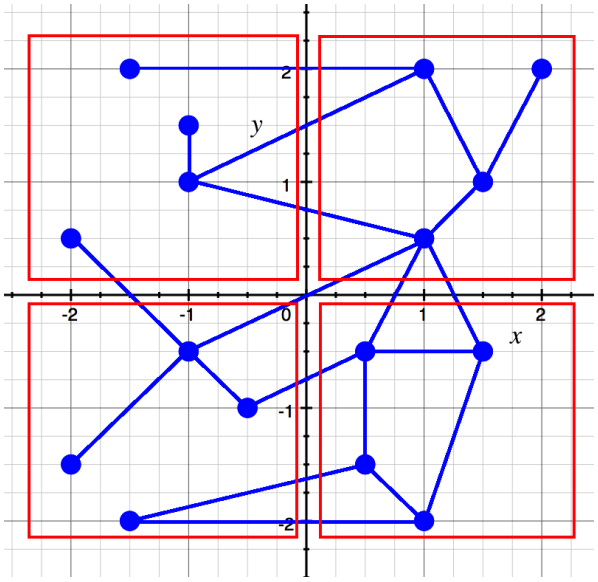


Figure 1.1: One of the problem of geometric partitioning algorithms is that the obtained partitions consist in disconnected nodes.

increases, geometric methods get slower. Lastly, they are confined to graphs whose nodes can be represented in the Euclidean space.

Later, works shifted towards spectral methods, that instead exploit the set of edges in the graph, but can be applied only on small structures due their time complexity. Spectral partitioning algorithms rely on one of the mathematical representations of a graph, the adjacency matrix, to study its connectivity properties and performing partitions accordingly to them. Although spectral methods results in optimal partitions, both in terms of size and number of edges crossing them, they are not scalable, becoming unfeasible to adopt even on medium sized graphs (thousands of nodes).

In the last decade, the majority of algorithms relied on multi-level heuristics, used to partition graphs of considerable dimension. The idea behind these approaches is straightforward and consists in three consecutive steps. First, a graph is reduced in size, aggregating nodes which satisfy a specific criterion: maximum weight of the edge connecting them, similar number of neighbours, etc. This step is repeated, coarsening the initial graph until a new graph with small enough size is obtained. Then, a partitioning algorithm is applied on the small graph, producing accurate results in short time. Lastly, the partitions are uncoarsened and refined, propagating the assignment of each node from the small to the initial graphs. These algorithms produce high quality outputs in very short times, having partitions that are almost perfectly balanced, outperforming geometric and spectral methods. They need, however, to be finely tuned in each step, considering also the possible correlations. For example, if the second phase is performed with an

algorithm that sacrifices the partitioning quality to obtain low computational time, the refinement should be strong and accurate, and vice versa.

Nowadays, the main goal is to develop algorithms increasingly scalable and fast, adopting parallelisation techniques, to move towards real-time applications.

1.1 THESIS CONTRIBUTION

We propose three new graph partitioning algorithms having different working principles and associated to distinct situations. The first algorithm, Label clustering, has the goal to obtain a graph partitioning based on nodes similarity, following a multi-level approach. A graph is first clustered aggregating nodes with the maximum similarity, using a modified version of the depth-first visiting on graphs, then it is reduced in size until a small enough structure is obtained, to apply a direct partitioning method.

The second algorithm, AD-tree partitioning, exploits a new data structure to perform a fast partitioning (linear in the number of nodes). The mentioned structure is the union of three elements: a directed graph with multiple connected components (named descendants graph), a normal tree derived from a graph and a directed tree (named Ariadne's tree), whose nodes are connected as if they were part of a chain (meaning that each node, except from the first and the last, has an edge pointing to only one other node and is pointed by only one incoming edge). The algorithm is based on value back-propagation and traversal of the AD-tree nodes, allowing to cut groups of nodes with a speed superior to the state of the art algorithms, independently from the number of desired partitions.

The last algorithm, called Directed partitioning, does not rely on any structure or data similarity, but works directly on the graph while exploring it. The idea behind it is to use a visiting procedure similar to depth-first, to obtain perfectly balanced partitions, having also high intra-connectivity. Different problems generate from the depth-first visiting, such as the formation of holes in the graph, mostly caused by the graph complexity. Directed partitioning solves these problems, at the cost of a slightly worse time performance with respect to the AD-tree partitioning algorithm, but providing perfectly balanced partitions.

Summing up, the three proposed algorithms contribute in different ways to the graph partitioning literature. The first partitions graphs with a hybrid multi-level approach, obtaining groups containing nodes with high similarity, in reasonable time (few seconds to cluster graphs with millions of nodes and more edges). The second exploits an enhanced data structure to obtain partitions in a time that is one order of magnitude less than the time taken by state of the art algorithms, sacrificing some balance in the partitions. The

third obtains perfectly balanced blocks, one of the most desired characteristics of a graph partitioning algorithm, in short time.

Moreover, the tree based partitioning grants that if a graph is connected, also the resulting partitions will have the same property, feature rarely showed in literature. The first algorithm also maintains the graph connected, but only for the reduction and coarsening phases: the direct partitioning algorithm on the smallest graph can, instead, produce partitions corresponding to disconnected components in the original graph. The same goes for Directed partitioning, that grants the property until all the holes are filled (including this phase).

1.2 THESIS OUTLINE

The thesis is organized in two parts. In the first, we provide the background materials needed to understand the contributions of the thesis.

- Chapter 2 introduces some mathematical concepts, such as definitions and theorems related to graphs and graph partitioning. Then, a review of the literature is presented, describing the evolution of the algorithms used for graph partitioning, focusing on some of them.

The second part contains the main contribution of this thesis, describing three new methods for graph partitioning.

- Chapter 3 illustrates the Label Clustering multi-level algorithm. After having defined new mathematical concepts, we focus on describing each phase of the algorithm. First, we consider the initial graph reduction, whose purpose is to build, from a graph, a new graph such that two adjacent nodes with the same label do not exist. Then, a simple coarsening scheme is proposed, to further decrease the size of the reduced graph. Lastly, we describe the method used to partition the smallest obtained graph.

- Chapter 4 presents a new data structure to represent graphs, the AD-tree, and a partitioning algorithm based on its traversal. The AD-tree is derived from three other structures, each described in detail: the tree corresponding to the graph; the graph formed by all the nodes and the edges not inserted in the tree, named descendants graph; the tree representing the order of insertion of the graph nodes in the first structure, named Ariadne's tree. After an overview of the AD-tree properties and methods of creation, the focus shifts on the partitioning algorithm, presenting and describing each routine that form it.

- Chapter 5 revisits the depth-first procedure on graphs, deriving a new algorithm, named Directed partitioning, that exploits the adjacency list

of the graph nodes to obtain perfectly balanced partitions having also regular shapes. In particular, we show some of the problems related to the use of depth-first for partitioning, solving and adapting them to the proposed algorithm.

- Chapter 6 is composed by two sections. In the first a specific use case is considered, where graph partitioning algorithms can be used to improve the state of the art results: 3D reconstruction. The section introduces, briefly, the reconstruction problem and the literature behind it, focusing on very recent algorithms. Then, we show how our graph partitioning methods can be used to reduce the time required to obtain accurate reconstructions.

  The second section includes three improvements of the AD-tree partitioning algorithm described in 4, all relying on the descendants graph. The first is a pre-processing technique, that improves the balance of the obtained data structure from the graph (the AD-tree). The second is a greedy heuristic used to support the partitioning algorithm increasing its cut possibilities. The last improvement is based on post-processing exchange of elements between partitions, once again to improve their overall balance.

Lastly, the thesis ends revisiting briefly the graph partitioning problem and the three algorithms, including possible improvements and future works.

Part I

LITERATURE BACKGROUND

2

GRAPH PARTITIONING

In this chapter the graph partitioning problem is presented, along with the common ideas behind the majority of algorithms developed to solve it. The chapter begins introducing known mathematical concepts and definitions about graphs. Then, the graph partitioning problem (GPP) is defined, followed by an overview of the algorithms and heuristics proposed over the last decades, distinguishing the approaches by purpose, output, performance and methodology. Lastly, some of the algorithms described briefly in the overview are discussed in detail, to show their working principle and implementation.

2.1 MATHEMATICAL BACKGROUND

The focus on the thesis will be mainly on undirected graphs, both weighted and unweighted. They are defined as:

Definition 2.1. Undirected weighted graph An undirected weighted graph $G(V, E, W)$ is a structure defined and characterized by:

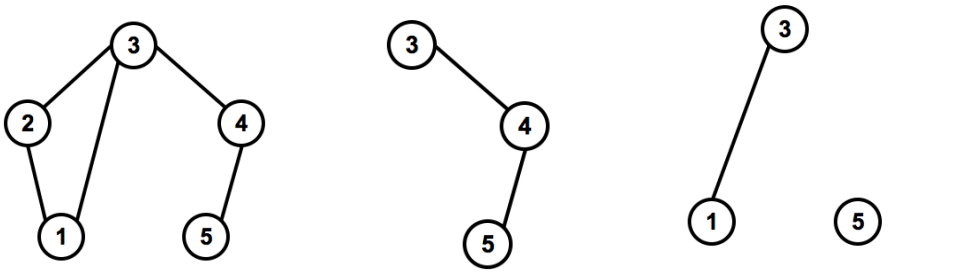
1. a set of nodes, or vertices, V
2. a set of edges between nodes $E \subseteq V \times V$, that represent the interaction of vertices
3. a function $W : E \to \mathfrak{R}$ that assigns a weight to the edges, named weight function

For each pair of nodes such that there is an edge between the first and the second one, there is also an edge connecting the second to the first node, making the graph undirected:

$$\forall v, u \in V \mid \exists e = (v, u) \in E \implies \exists e' = (u, v) \in E$$

Definition 2.2. An undirected graph (unweighted) $G(V, E)$ is an undirected weighted graph for which the weight function is constant equal to 1.

$$W : E \to 1$$



(a) Graph **(b)** Subgraph **(c)** Subgraph with disconnected nodes

Figure 2.1: A graph with two possible subgraphs.

The first two elements of a graph are the fundamental parts: a set of nodes and a set of edges that describe the relations between nodes. The third element makes a graph weighted, associating each edge to a real number. Weights can express different information, such as similarity, degree of connectivity, distance, cost, capacity, etc.

A graph can be considered in its entirety or just within a subset of it, called induced subgraph and defined as:

Definition 2.3. Induced subgraph Let G be a graph and let $S \subseteq V$ be a subset of the graph nodes. The induced subgraph G_S is the graph whose vertex set is S and whose edge are all the edges of E that have both extremes in S. If the graph is weighted, the edges in G_S have the same weights as in the original graph G.

From now on, instead of using the term induced subgraph, only subgraph will be used. Since the conditions on the subgraph are only relative to the node and edge sets, one should also consider extreme cases: a graph is always a subgraph of itself, because it is defined over all of its nodes. Moreover, a subgraph can have nodes not necessarily connected by edges, as showed in Figure 2.1.

Each edge allows a node to interact with other vertices, that form its so called neighbourhood, or adjacency set, defined as:

Definition 2.4. Adjacency set Given a node v, all the vertices connected to it by edges are called adjacent nodes and are defined as the set

$$N_G(v) = \{u : (v,u) \in E\}, \{v,u\} \in V$$

called adjacency set.

This is the first step towards a non-graphical representation of the graph, that gives local information for every node. Considering Figure 2.1a, node 1 has 2 and 3 as neighbours; node 4 is adjacent to both nodes 3 and 5, and

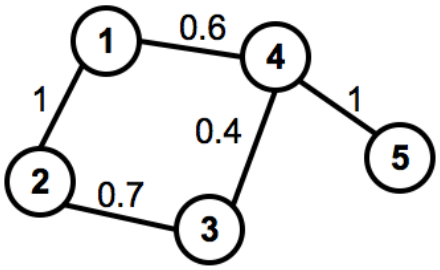


Figure 2.2: A small undirected weighted graph.

so on. The adjacency concept defines also the degree of a node, representing the sum of all the weights associated to the edges that have the node as one extreme.

Definition 2.5. Degree of a node The degree of a node v_i, indicated as $d(v_i)$, is the sum of all the weights of its incident edges:

$$d(v_i) = \sum_{v_j:(v_i,v_j)\in E} W(v_i, v_j)$$

If a graph is unweighted, the degree of a node is the number of adjacent vertices.

A graph can be represented it in different ways. Following, two of the most important representations are reported:

- an adjacency list, that is a collection of unordered lists that describe the set of neighbours of a node
- an adjacency matrix, that is a matrix of size $n \times n$, where $n = |V|$, where a cell of coordinates (i, j) contains the weight associated to edge (v_i, v_j) (0 if no edge exists)

An example is given in Tables 2.1a and 2.1b, referred to the graph in Figure 2.2. It is important to notice that the adjacency list is represented in the following way: v | ... v_i^{adj} $W(v, v_i^{adj})$ ... This means that a node is followed by the list of adjacent nodes, each with the weight of the edge corresponding to them. It is intuitive that the matrix representation is preferred when there are lots of edges (dense graphs), while the list is suggested when the number of connections is limited (sparse graphs). The reason is that while the matrix representation has quadratic space complexity with respect to the number of nodes and the list has linear or, in the worst case, quadratic space requirements, a search would be done in a constant time in the matrix but linearly, in the worst case, in the list.

$v_i \backslash v_j$	1	2	3	4	5
1	0	1	0	0.6	0
2	1	0	0.7	0	0
3	0	0.7	0	0.4	0
4	0.6	0	0.4	0	1
5	0	0	0	1	0

(a) Adjacency matrix

v_i				
1	2	1	4	0.6
2	1	1	3	0.7
3	2	0.7	4	0.4
4	1	0.6	5	1
5	4	1		

(b) Adjacency list

Table 2.1: Different representations of the same graph, showed in Figure 2.2.

The adjacency matrix can be defined in in a more suitable way:

Definition 2.6. Adjacency matrix Let $G(V, E, W)$ be a weighted undirected graph, with weight function $W : E \to \mathfrak{R}$. We define the adjacency matrix $A_G \in \mathfrak{R}^{V \times V}$ as follows:

$$A_{i,j} = \begin{cases} 0 & \text{if } i = j \\ W(i,j) & \text{otherwise} \end{cases}$$

Matrix A has multiple properties. It is symmetric, because of the undirected property of the edges, and from it one can recognize nodes with no connections, that correspond to rows (or columns) containing only zeros. From the adjacency matrix one can also detect sequences of nodes and edges moving alternately horizontally and vertically. Looking at the adjacency matrix in Table 2.1 and starting from node 1, an example can be constructed to obtain multiple, but not all, the sequences in the associated graphs.

1. From 1 one moves horizontally until a connected node is found. 2 is encountered, so one starts moving downward, until node 3 is met. Going to the left does not add any node to the sequence, while going to the right node 4 is found. Stop here, the sequence 1-2-3-4 is obtained.

2. From 1 one moves horizontally and surpasses node 2 until node 4 is reached. From here one goes down towards node 5, that is a dead end. The obtained sequence is 1-4-5.

Before, it was stated that each node has a degree dependant on its neighbours (Definition 2.5). As adjacent nodes can be represented with a matrix, also degrees can be expressed in a matricial form, in the following way:

Definition 2.7. Degree matrix Let $G(V, E, W)$ be a weighted undirected graph, with weight function $W : E \to \mathfrak{R}$. The degree matrix $D_G \in \mathfrak{R}^{V \times V}$, of G, is computed as:

$$D_{i,j} = \begin{cases} d(v_i) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

where $d(v_i)$ has been defined at 2.5.

It is easy to associate the adjacency and degree matrices. Because $d(v_i)$ is the sum of all the weights of the incident edges of v_i, then it is also true that $d(v_i)$ is the sum of the values in the i^{th} row of the adjacency matrix.

The sequences of nodes computed during the last part of the explanation regarding the adjacency matrix are called paths, and are defined in the following way:

Definition 2.8. Path A path in a graph $G(V, E)$ is a sequence of alternated nodes and edges v_i, e_i, v_{i+1}, ..., e_{j-1}, v_j such that:

- $v_i, v_{i+1}, ..., v_j \in V$
- $e_i, e_{i+1}, ..., e_{j-1} \in E$
- e_i is the edge connecting v_i and v_{i+1}

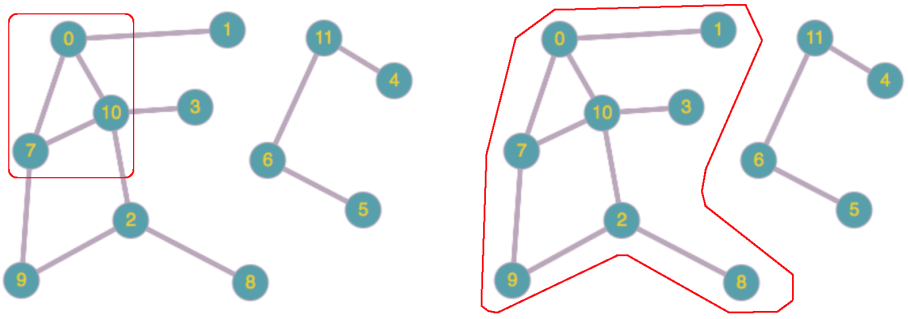
Paths in a graph indicate its density and degree of connectivity. For example, a large number of edges allows to find an incredibly large amount of distinct paths. Certain areas in graphs are characterized by all of their nodes being connected by one or more paths. These areas are defined as connected subgraphs.

Definition 2.9. Connected subgraph Given a graph $G(V, E)$, a subgraph G_S of G, defined over vertex set $S \subset V$, is connected if for every pair of distinct vertices in S there is a path in $G_S(S, E')$, with $E' \subseteq E$:

$$\forall s_i, s_j \in S, i \neq j, \exists (s_i, e'_i, s_{i+1}, ..., e'_{j-1}, s_j) \mid s_i, ...s_j \in S, e'_i, ...e'_{j-1} \in E'$$

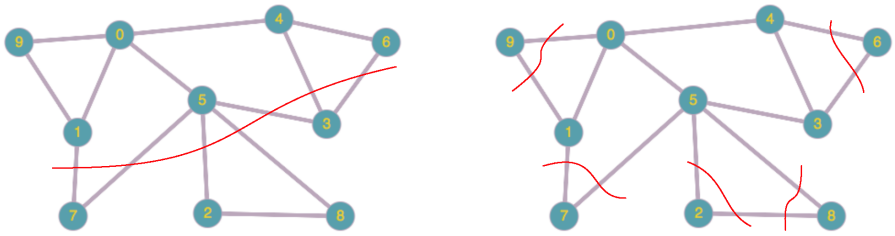
Definition 2.10. Connected component A subgraph G_S of graph $G(V, E)$ is maximally connected if it is connected and for all the vertices $v \in V$ such that $v \notin S$, there is no other node $u \in S$ for which edge $(v, u) \in E$. A maximally connected subgraph is also known as connected component.

An example of these subgraphs is given in Figure 2.3a. Connected components form stand-alone subgraphs, which share no interaction among each other. This is important when analysing large and complex graphs because it reduces the computational resources used when performing different actions on them such as researching a node, ordering or partitioning. The detection of connected components is a well known problem and it has been



(a) A connected subgraph **(b)** A maximally connected subgraph

Figure 2.3: The Figure on the left represents a possible connected subgraph, that is not maximal. On the right, a connected component is showed.



(a) Graph with a generic cut **(b)** Graph with minimum cuts

Figure 2.4: On the left, the graph is divided by a cut of weight 7; on the right the same graph is divided by multiple cuts, that are all minimum cuts, of weight 2.

intensively studied since the last century. A remarkable variety of algorithms is available to find the connected components in an undirected graph: from serial to parallel approaches, from small to large scaled graphs, as in [6–10].

The last concept seen in this background is the cut.

Definition 2.11. Cut A cut is defined as a pair $(S, V \setminus S)$, with $S \subseteq V$. A cut is characterized by a **weight**, that is the sum of all the weights of the edges with one extreme in S and the other in $V \setminus S$.

The set of edges that cross the partitions is said cut-set. A cut is minimum if its weight is the lowest between all the possible cuts on the graph; it is also not true that, given a graph, the minimum cut is unique (as in Figure 2.4).

2.2 THE GRAPH PARTITIONING PROBLEM (GPP)

One of the main problems that is associated to graphs is how to partition them. A graph can represent any set of objects that interact in a specific way: the objects are the nodes and their relations are represented as edges. Graph partitioning is a way to distribute nodes and relations, to process them efficiently for any possible application. Some examples of real-world problems

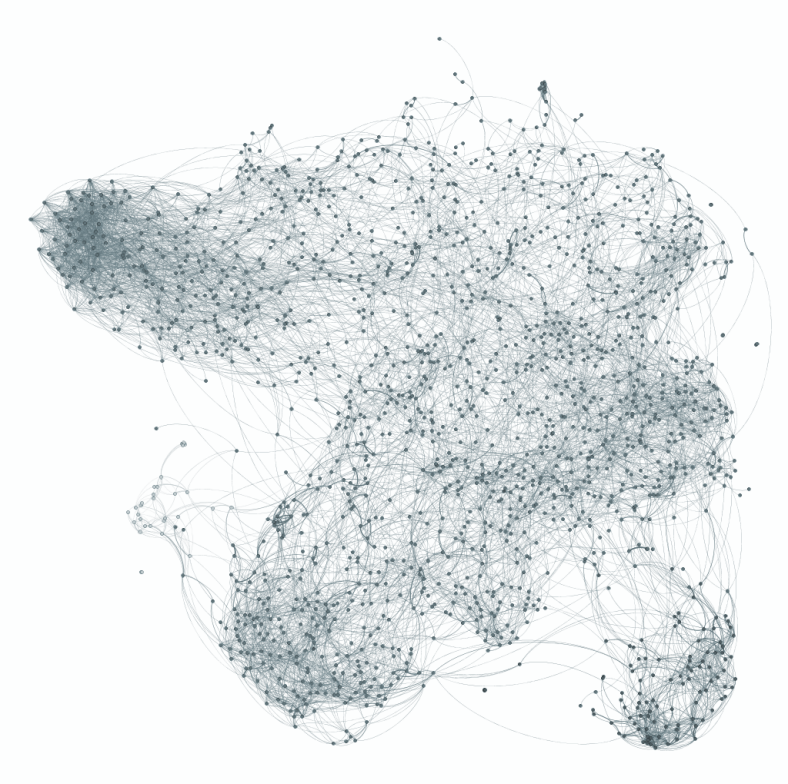


Figure 2.5: A complex graph.

that can be represented using graphs are related to networks (transportation, biological, social), image processing, security management [11], etc. Partitioning is necessary when the amount of data to consider is too big (example showed in Figure 2.5) to be analysed with single devices due to the computational resources needed, and should instead be distributed between multiple communicating workstations, that can operate on smaller subgraphs.

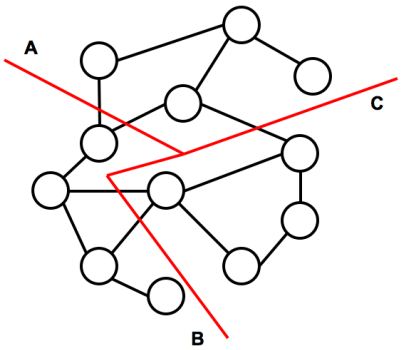
The partitioning problem can be formally defined in the following way:

Definition 2.12. Graph partitioning problem Given a natural number k, greater than 1, and an undirected graph with non-negative weights $G(V, E, W)$, the graph partitioning problem is the task to divide G into k subsets (partitions) $V_1, V_2, ..., V_k \subset V$, such that:

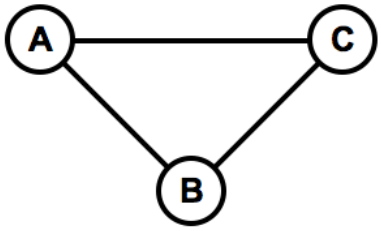
- $V_1 \cup V_2 \cup ... \cup V_k = V$
- $V_i \cap V_j = \emptyset, \forall i, j \in \{1, ..., |V|\}$

The problem is also known as k-way partitioning.

Each subset of nodes V_i defines also a subgraph $G_{S_i}(V_i, E_i, W_i)$ with the following properties:



(a) Partitioned graph



(b) Quotient graph

Figure 2.6: A graph partitioned into three blocks, of four elements each, and its quotient graph.

- $V_i \subset V$
- $E_i \subset E$
- $W_i(u,v) = \begin{cases} W(u,v) & \text{if } u,v \in V_i \\ 0 & \text{otherwise} \end{cases}$

The partitioning problem can be seen as maximisation or minimisation of certain objective functions, and can be constrained in different ways. The most known constraint that can be added is to have balanced partitions:

Definition 2.13. Balanced graph partitioning A balanced graph partitioning problem is a partitioning problem that must respect a balance constraint, demanding that all the subgraphs obtained have about the same number of nodes. In particular it requires that

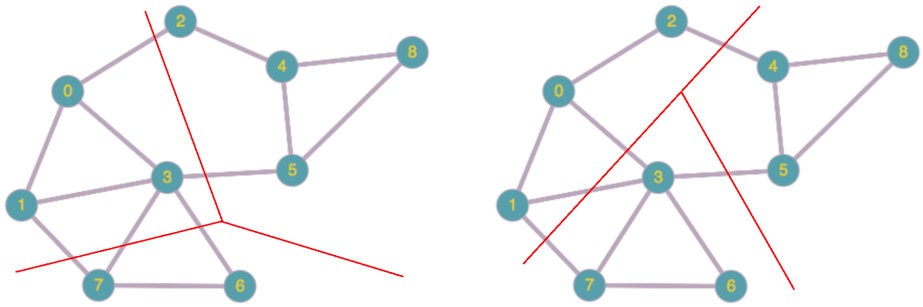
$$\forall i \in 1,...,k, |V_i| \leqslant (1+\epsilon)\lceil |V|/k \rceil$$

for some **imbalance parameter** $\epsilon \in \mathfrak{R}_{\geqslant 0}$.

To evaluate the quality of a partitioning, one can compute the maximum imbalance of the partition, computed as:

$$max_i |V_i|/\lceil |V|/k \rceil$$

From the partitions a new graph can be built, called quotient graph, having k nodes such that node $i \in 1,...,k$ corresponds to the node set V_i. An edge between nodes in the quotient graph exist only if there is an edge that runs between the node sets in the original partitioned graph (as in Figure 2.6). A particular case of balanced partitioning is given by $\epsilon = 0$, meaning that all



(a) Balanced partition **(b)** Perfectly balanced partition

Figure 2.7: The left Figure represents one of the possible balanced partitions, while on the right a perfectly balanced partition is showed.

the partitions contain the same amount of nodes. This is also known as perfectly balanced partitioning, and is a well studied problem in the scientific community, especially from a theoretical point of view.

Two fundamental theorems describe the degree of complexity of the perfectly balanced partitioning.

Theorem 2.1. (Garey, Johnson, Stockmeyer) Perfectly balanced bipartitioning is NP-complete [12].

Theorem 2.2. (Andreev and Räcke) For $k \geqslant 3$ the perfectly balanced partitioning problem has no polynomial time approximation algorithm with finite approximation factor unless P = NP [13].

Due to the fact that perfectly balanced partitioning is hard to solve exactly, works in literature rely on heuristics and greedy algorithms that provide non-optimal solutions, being satisfied with an outcome associated to $\epsilon > 0$ (so using a relaxation of the perfectly balanced constraint). Following these approaches two kinds of partitions are obtained: $|V_i| < (1+\epsilon)\lceil |V|/k \rceil$ and $|V_i| > (1+\epsilon)\lceil |V|/k \rceil$, respectively called underloaded and overloaded partitions. Computational time is another metric used to evaluate an algorithm for graph partitioning.

The example in Figure 2.7 can be used to summarize the concepts just seen: the left graph is partitioned into 3 subgraphs and has a maximum imbalance of 1.333 (meaning that $\epsilon \geqslant 0.333$), while the right one is perfectly balanced and each group has the same number of nodes.

After the brief introduction of the most important constraint that can be applied to the graph partitioning problem, the focus can now shift on some of the objective functions associated to it:

- minimisation of the total cut
- maximisation of the minimum sum of weights inside a partition

The first objective function is used when a communication or transportation problem is formulated with a graph, or simply when partitions isolated with respect to each other are desired. The second objective function is instead used when communities and common patterns in the data, represented as a graph, have to be detected. The latter is usually associated to the clustering problem, but differently from it, it requires k partitions, while clustering has no balance constraint and no fixed number k (it dynamically changes during the algorithms iterations).

2.3 THE CLUSTERING PROBLEM

Although in this thesis the main focus will be on the GPP, it is also useful to know how partitioning differs from clustering to proper understand one of the proposed algorithms.

Clustering a graph still means partitioning it, but with the differences that k is not fixed and there is no constraint over the size of the partitions. Clustering is needed to group together nodes that share common traits and to identify communities inside the data represented using a graph. For example, one could want to bring together all the nodes that have at least n neighbours, or that are connected to other nodes with edges that have a weight above a certain threshold α.

A small example is given in Figure 2.8. There, a set of planar points is represented as a fully connected graph such that:

- a node is characterized by two coordinates $i, j \in \mathfrak{R}$
- the weight of the edges is computed as the 2D Euclidean distance between the nodes

Suppose that the main goal is to cluster the graph such that all the edges inside a group of nodes have value < 1, meaning elements with small distances should be together. What can be obtained is represented in Figure 2.8b.

2.4 GRAPH PARTITIONING ALGORITHMS IN LITERATURE

The main ideas behind the algorithms used to solve the graph partitioning problem can be distinguished between graphs nodes associated to geometric coordinates and graphs without them. Some methods work on entire graphs and can compute a solution directly. These algorithms have been recently used as subroutines of more complex procedures to add scalability and perform partitioning on larger structures.

Few graphs are provided with n-dimensional coordinates, due to the data that they are representing: meshes, elements with features that can be discretized in $\mathfrak{R}^n$, etc. This peculiar category of graphs can be partitioned us-

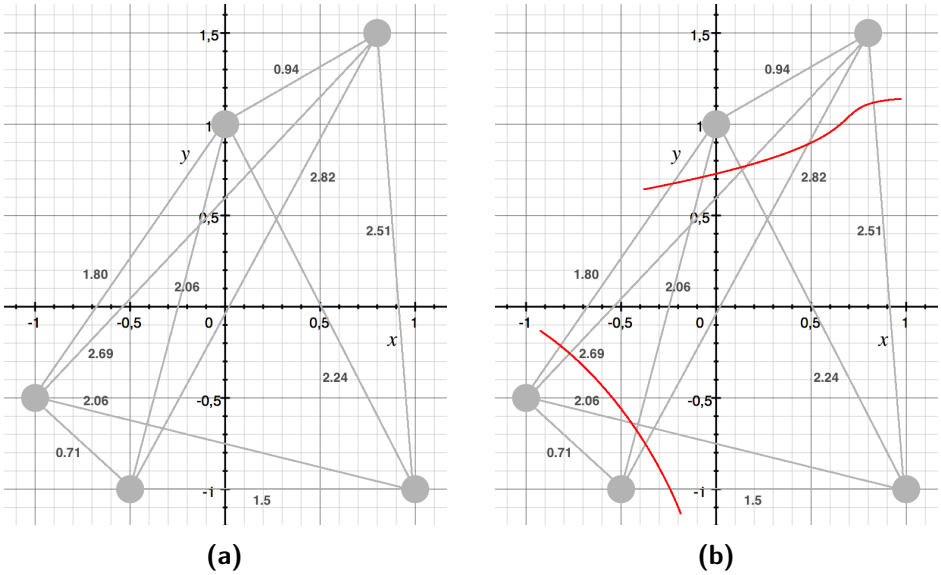


Figure 2.8: Figure 2.8a is the graph representation of some 2D points, considering their Euclidean distance as weight for the edges. To the right, in Figure 2.8b, there is the clustering obtained imposing a maximum distance between nodes of 1.

ing the associated geometry, so the first topic will be about algorithms that partition spaces in which the graphs are embedded. Next, some advanced concepts about the adjacency matrix, introduced in Definition 2.6, will be deepened, seeing how a different category of algorithms is derived from it. Lastly, the focus will shift on the commonly used approaches for graph partitioning: multi-level algorithms that work on a graph reducing its size to partition a smaller version of it, trying to maintain its connectivity properties.

2.4.1 *Geometry based graph partitioning*

Graphs can be associated with geometric concepts, such as coordinates. For example, a 3D polyhedral surface can be viewed as a graph having as many nodes as the number of the surface vertices, with edges corresponding to the edges of the multidimensional structure (Figure 2.9) Partitioning using nodal coordinates is a well known problem in the old literature. Some of the most famous algorithms are recursive coordinate bisection (RCB) [14] and inertial partitioning [15, 16].

RCB is a recursive algorithm that at each step projects the graph nodes onto the coordinate axis with the longest domain range (x, y or z direction). The nodes are then separated into two groups, using the median of their projections, through a bisecting plane (or line, if we consider the 2D case) that is orthogonal to the considered coordinate axis. It is a divide and conquer

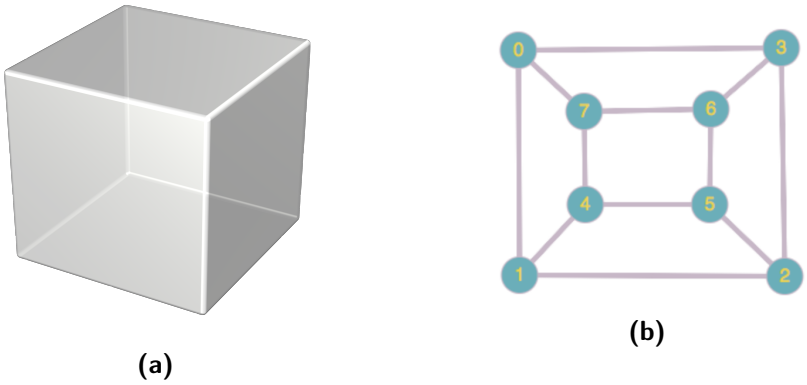


Figure 2.9: A cube and the graph representing it.

approach and has the interesting property that can be applied to any data with features that can be discretised. Think, for example, to a representation of a social network where each node is a person characterized by certain features (height, age, grade of education and so on): they can be represented as a graph with nodal coordinates, associating a value to each feature. The graph can be then partitioned using RCB in a higher dimensional case (number of the features). Although the implementation is straightforward and easy, RCB may output disconnected partitions. An example of how RCB works is given by Figure 2.10.

Inertial partitioning can be considered an improvement of RCB in terms of worst case performance, because its bisecting plane is orthogonal to a plane L that minimizes the moments of inertia of nodes. In other words, the projection plane L is chosen such that it minimizes the sum of squared distances to all nodes. Inertial partitioning retains the simplicity of RCB and improves the quality of the partitioning, but shows the same problems: since the graph connectivity is not taken into account, it may produce disconnected subgraphs.

Both RCB and inertial partitioning separate nodes according to their projected position with respect to computed hyperplanes, limiting the partitioning quality in terms of partitions connectivity and resulting shapes: nodes are divided into rectangular blocks, without considering if there is a more suitable geometric structure to represent them. Miller et al. further improved RCB, proposing the random spheres algorithm [17, 18], summed up by the following six steps, showed in Figure 2.11.

1. The nodes are projected stereographically from $\mathfrak{R}^d$ to the unit sphere centred at the origin in $\mathfrak{R}^{d+1}$ (Figures 2.11a,2.11b and 2.11c).

2. A centre point of the projected nodes in $\mathfrak{R}^{d+1}$ is found.

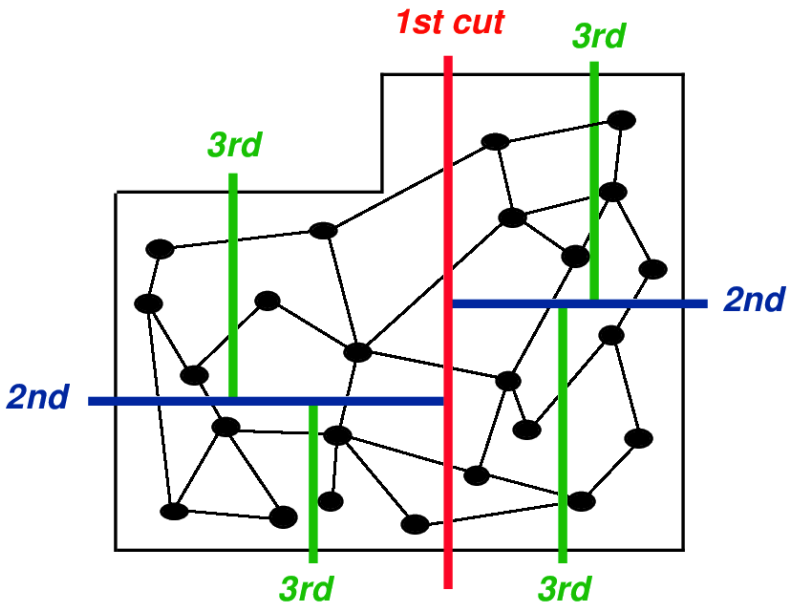


Figure 2.10: Example of how a planar graph is partitioned by the RCB algorithm.

3. The projected nodes are rotated around the origin in $\mathfrak{R}^{d+1}$ so that the centre point has new coordinates $(0, 0, ..., 0, r)$ on the $d+1^{st}$ axis. Then the nodes are dilated on the surface of the sphere so that the centre point becomes the origin.

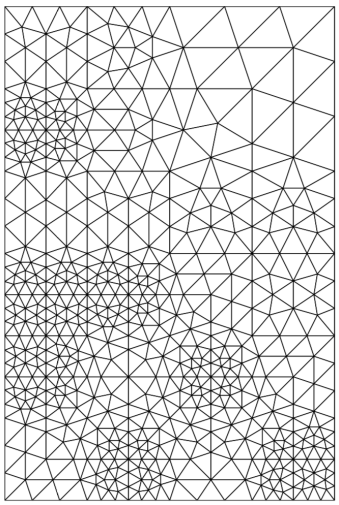
4. A random great circle is chosen (a d-dimensional unit sphere) on the unit sphere in $\mathfrak{R}^{d+1}$ (Figure 2.11d).

5. The great circle is then transformed to a circle in $\mathfrak{R}^{d}$, undoing the stereographic projection (Figure 2.11e).

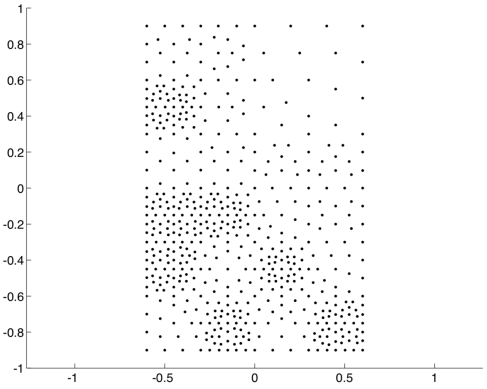
6. Lastly, the circle is converted into a separator, that is a small set of vertices that divides the graph in half (Figure 2.11a).

A different approach is given by using space-filling curves: continuous curves which completely fill up higher dimensional unit hypercubes [19, 20] (example in Figure 2.12). Due to their fractal nature, they possess locality preserving properties: consecutive elements in the curve-induced order tend to lie close to each other in space, and vice versa. Although space-filling curves were originally defined on grids, later works [21] demonstrated that they can also be used to partition graphs. The graphs nodes are separated recursively and aligned according to the underlying curve structure. The recursion is repeated until each subpart contains a single vertex.

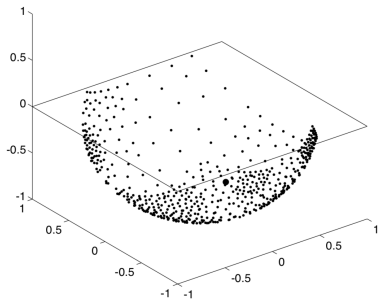
All of these algorithms have a drawback: they do not exploit the connectivity of a graph. For this reasons, the focus shifted towards new approaches: either the quality of the partitioning is improved (execution time and shape of



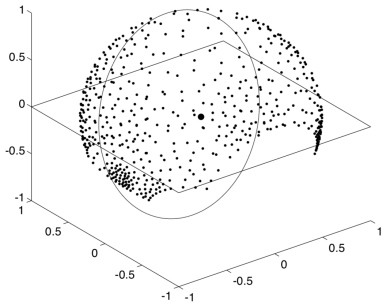
(a) Graph to partition



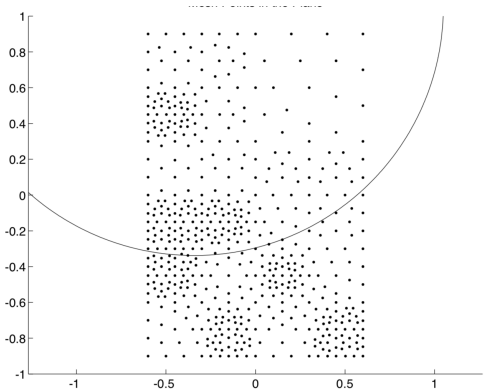
(b) Geometric representation



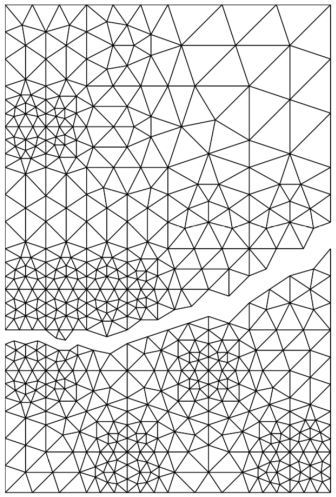
(c) Stereographic projection



(d) Great circle selection



(e) Nodes re-projection



(f) Partitioned graph

Figure 2.11: Example of how the random spheres algorithm works.

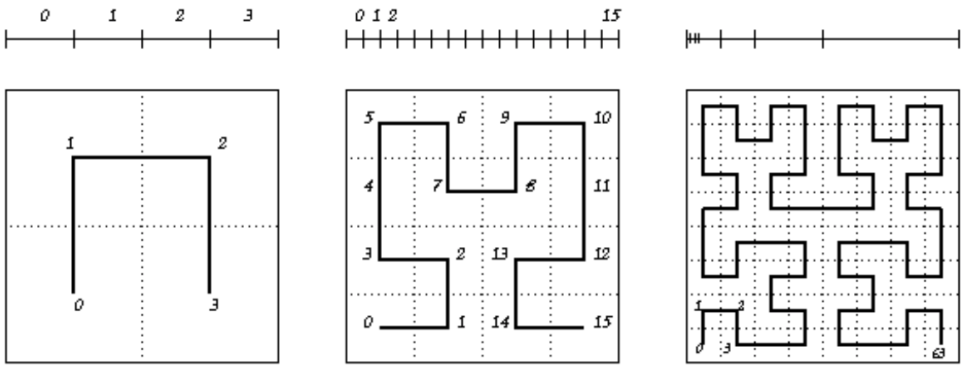


Figure 2.12: Example of a space filling curve, increasing in precision and complexity.

the resulting partitions), exploiting distributed algorithms, or the geometric information is embedded into other types of graph partitioning algorithms [22–24].

2.4.2 *Spectral partitioning*

Some concepts are needed to describe a different kind of algorithms used for partitioning, based on spectral information of a graph.

Definition 2.14. Eigenvector and Eigenvalue An eigenvector of a square matrix $A \in \mathfrak{R}^{n\times n}$ is a non-zero vector $v \in \mathfrak{R}^n$ that, when A is multiplied by v, yields the constant multiple of v.

$$Av = \lambda v$$

The number $\lambda \in \mathfrak{R}$ is called eigenvalue of A corresponding to the eigenvector v.

Certain matrices have special features concerning eigenvectors and eigenvalues, as the positive semi-definite matrices, defined as:

Definition 2.15. Positive semi-definite matrix Let $A \in \mathfrak{R}^{n\times n}$ be a symmetric matrix. It is positive semi-definite if it verifies one of the following properties:

- $\forall x \in \mathfrak{R}^n_{\neq 0} : x^T Ax \geqslant 0$
- all eigenvalues of A are $\geqslant 0$

From the adjacency and degree matrices, one can build a new matrix L, called Laplacian matrix of a graph.

Definition 2.16. (unnormalized) Laplacian Matrix Let $G(V, E)$ be a graph. The matrix $L = D - A$, where D is the degree matrix and A is the adjacency matrix, is called (unnormalized) Laplacian matrix of G.

The Laplacian matrix has some important properties in the case of undirected graphs.

Theorem 2.3. The Laplacian matrix L of a graph G is symmetric if and only if G is undirected.

Proof. The sum, difference and element-wise multiplication of two matrices are symmetric if both are symmetric. D is a diagonal matrix and consequently is symmetric. A is symmetric only if representing unweighted graphs (because each edge appears twice), hence L is symmetric only if the associated graph is undirected. ■

Theorem 2.4. The Laplacian matrix L is positive semi-definite.

Corollary 2.5. *There is a trivial eigenvector* $\mathbb{1} = (1, \dots, 1)^T$ *of the graphs Laplacian. Its eigenvalue is* 0.

The results of these theorems can be showed on the graph of Figure 2.2, whose adjacency matrix is in Table 2.1:

$$L \cdot \mathbb{1} = \begin{bmatrix} 1.6 & -1 & 0 & -0.6 & 0 \\ -1 & 0.7 & -0.7 & 0 & 0 \\ 0 & -0.7 & 1.1 & -0.4 & 0 \\ -0.6 & 0 & -0.4 & 2 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = 0 \cdot \mathbb{1}$$

Lastly, another important theorem is enunciated (but not proved).

Theorem 2.6. The number of occurrences of the eigenvalue 0 in the set of eigenvalues, known as spectrum, of the graphs Laplacian is the number of connected components.

The partitioning problem can now be restated as follows: given an undirected weighted graph that represents some data and whose weights indicate the similarity between them, the graphs has to be partitioned such that the edges between different partitions have a very low weight (which means that separated nodes are dissimilar from each other) and the edges within a group have high weight (which means that nodes within the same partition are similar to each other). Two objective function can be associated to this problem:

- minimisation of the RatioCut [25], that is the sum of the cuts normalised with respect to the size of each partition

$$\mathrm{RatioCut}(V_1, V_2, \ldots, V_k) = \sum_{i=1}^{k} \frac{\mathrm{cut}(V_i, \overline{V_i})}{|V_i|}$$

- minimisation of the Ncut [26], that is the sum of the cuts normalised with respect to the volume of each partition, that is the sum of the weights of its edges

$$\mathrm{Ncut}(V_1, V_2, \ldots, V_k) = \sum_{i=1}^{k} \frac{\mathrm{cut}(V_i, \overline{V_i})}{\mathrm{vol}(V_i)}$$

where the cut between two sets of nodes has been defined at 2.11. The relaxation [27] of these problems lead to multiple algorithms that involve the manipulation of the Laplacian matrix of a graph: unnormalized spectral clustering if we relax RatioCut and normalized spectral clustering if we relax Ncut.

The first, unnormalized spectral clustering, is described as follows. Given the adjacency matrix $A \in \mathfrak{R}^{n \times n}$ of a graph G, the nodes can be partitioned into k groups with the following steps:

1. the unnormalized Laplacian L (Definition 2.16) of G is computed
2. the first k eigenvectors $v_1, \ldots, v_k$ of L, in ascending order by value, are derived
3. the matrix $V \in \mathfrak{R}^{n \times k}$, containing the previously computed eigenvectors as columns, is built
4. the rows of V, that are $y_i \in \mathfrak{R}^k, i \in \{1, 2, \ldots, n\}$ are taken singularly
5. the points y_i are clustered with the k-mean algorithm into blocks $C_1, \ldots, C_k$

Normalised spectral clustering is very similar and changes only the Laplacian considered [26]:

1. the unnormalized Laplacian L (Figure 2.16) of G is computed
2. the first k eigenvectors $v_1, \ldots, v_k$ of the generalized eigenproblem $Lv = \lambda Dv$, in ascending order by value, are derived
3. the matrix $V \in \mathfrak{R}^{n \times k}$, containing the previously computed eigenvectors as columns, is build
4. the rows of V, that are $y_i \in \mathfrak{R}^k, i \in \{1, 2, \ldots, n\}$, are taken singularly

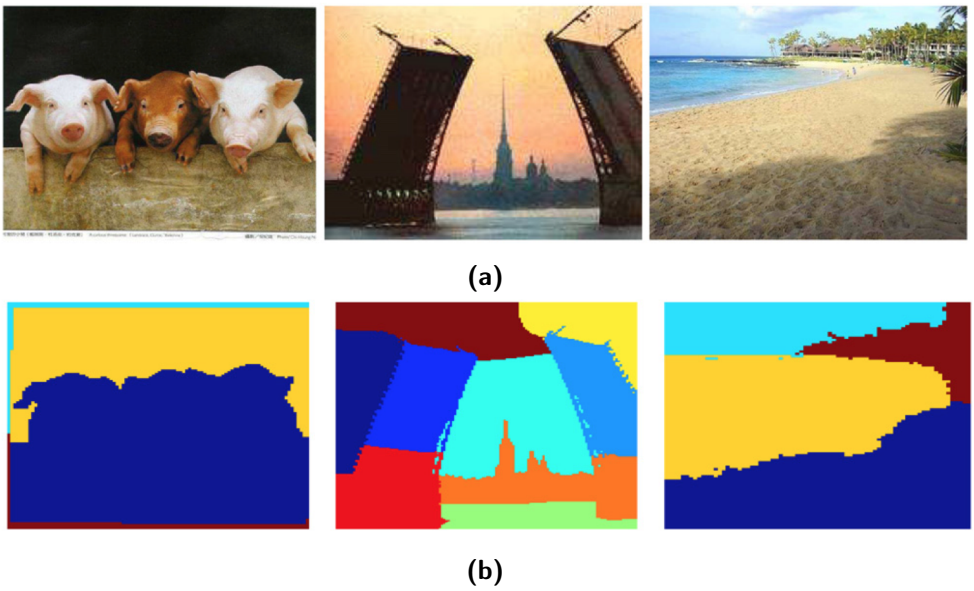


Figure 2.13: Example of image segmentation using the self tuning spectral algorithm proposed in [29].

5. the points y_i are clustered with the k-mean algorithm into blocks $C_1, \ldots, C_k$

where the eigenproblem $Lv = \lambda Dv$ correspond to the normalised Laplacian $L = I - D^{-1}A$. For both algorithms, once the clusters $C_1, \ldots, C_k$ are computed, one can obtain the desired partitions over the set of nodes of the graph, as $V_1, \ldots, V_k$ with $V_i = \{j | y_j \in C_i\}$.

The main advantage of spectral clustering techniques applied to graph partitioning is that they are simple to implement and can be solved efficiently by standard linear algebra methods, performing better than traditional clustering algorithms. However, they cannot be considered good solutions of the balanced partitioning problem (and in fact they are referred to as clustering techniques), since they rely on algorithms that do not care about the size of the partitions. Moreover, considering n nodes with f features/coordinates it is trivial to see that spectral clustering algorithms are indifferent to the number of features, while they suffer as n increases (they involve the manipulation of $n \times n$ matrices).

Later, scientific literature focused on solving the problems, described in [28], of spectral clustering, especially scalability, locality of information, eigenvector computation, etc. Zelink et al. [29] proposed an algorithm, named Self-Tuning Spectral Clustering, that automatically detects k, moving even further from the partitioning problem, going instead towards the clustering problem (example in Figure 2.13a).

In the last decade particular attention was directed towards the lack of scalability of standard spectral clustering algorithms: in [30] the issue is faced investigating two representative ways of approximating the dense similarity matrix to distribute the computation on multiple devices, obtaining a promising speed-up; in [31], instead, a large-scale multi-view spectral clustering approach is proposed based on the approximation of similarity graphs using bipartite graphs. Other works, such as [32] reduce the computational time adopting results in the emerging field of graph signal processing: graph filtering of random signals and random sampling of bandlimited graph signals.

Scalability and generalization are faced also adopting deep learning approaches to spectral clustering. In the recent SpectralNet network [33], a map that embeds input data points into the eigenspace of their associated graph Laplacian matrix is learned, followed by a standard clustering. SpectralNet is trained using a procedure that involves constrained stochastic optimization, that allows it to scale to large datasets. Moreover, the map learned by SpectralNet naturally generalizes the spectral embedding to unseen data points, allowing a dynamic modification of the original graph, feature that is not present in other algorithms.

2.4.3 *Multi-level graph partitioning*

To improve spectral bisection, that is spectral clustering with $k = 2$, a new idea for partitioning graphs was introduced in [34]: multi-level graph partitioning. The motivation behind multi-level approaches is that the partitioning is easier and more efficient to perform on smaller graphs, hence the need to find a way to shrink complex networks.

These coarse-to-fine algorithms consist of three steps. First, from the original graph, smaller and smaller graphs are created, in the so called coarsening phase. Second, the smallest graph in the sequence is partitioned, having particular attention to obtain balanced groups. Lastly, the partition is back-propagated, expanding the nodes of the smaller graph until the original structure is reached, including possible refinements over the partitions. The last phase is called graph uncoarsening with refinement (see Figure 2.14).

2.4.3.1 *Coarsening*

The coarsening phase is obtained following reduction (or contraction) schemes: edges are aggregated and replaced with a single weighted node. In this phase, a hierarchy of graphs with decreasing number of nodes is created in multiple ways.

One of the simplest and mostly used contraction scheme is the strict aggregation SAG [35, 36, 64, 37] (also known as edge contraction), where nodes

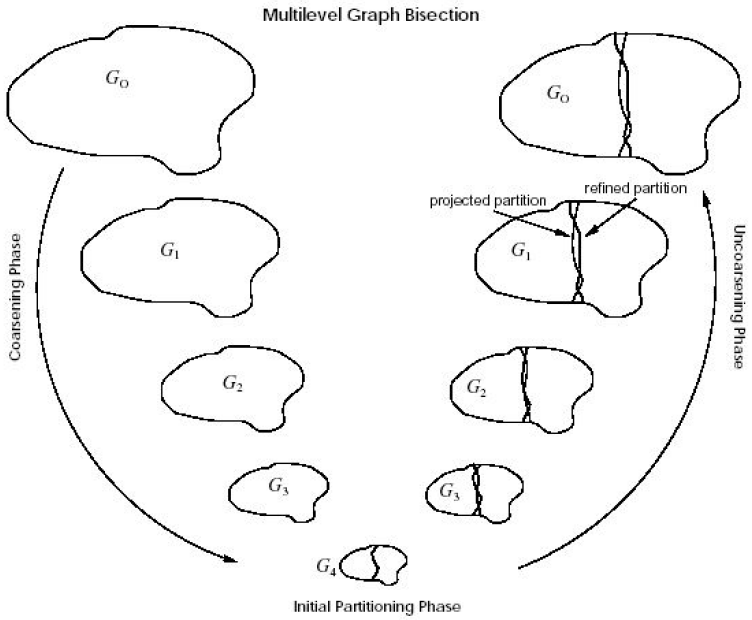


Figure 2.14: Multi-level partitioning scheme for bisection.

are inserted into small disjoint subsets, called aggregates. Two nodes v_i and v_j belong to the same aggregate if their coupling is locally strong, meaning that the weight of the edge connecting them is comparable to the maximum weight of their incident edges, in particular to

$$\min\{\max_k W(v_i, v_k), \max_k W(v_k, v_j)\}$$

An example is showed in Figure 2.15. Some of the known algorithms based on this scheme are presented and explained in the following list.

1. HEM (heavy edge matching) [51]: this algorithm matches every unmatched node with the free neighbours that share with it the maximum weight, considering the nodes in random order. Although this method is easy to implement and fast (linear in the number of edges), it does not guarantee high quality [36].

2. SHEM (sorted HEM): used and presented in the famous METIS partitioner [38], it matches nodes in the same way as HEM, but considers them in non-decreasing order of degree, randomizing in case of degree equality. It provides better results, with the same time complexity as HEM.

3. GEM (greedy edge matching) [41]: edges are ordered according to their weights, in non-increasing order, and scanned from the first to the last.

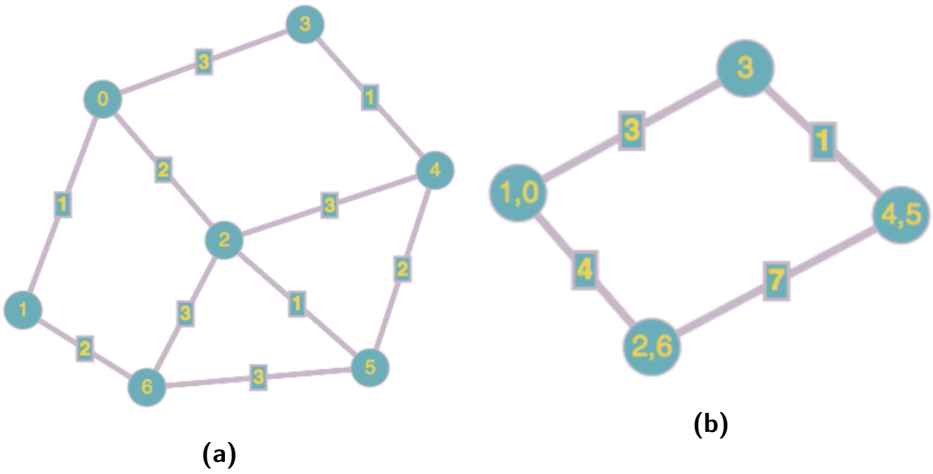


Figure 2.15: Example of a possible implementation of the SAG scheme; notice how the reduction obtained, starting from the random node ordering {2,4,1,5,3,0,6}, is not the best reduction possible.

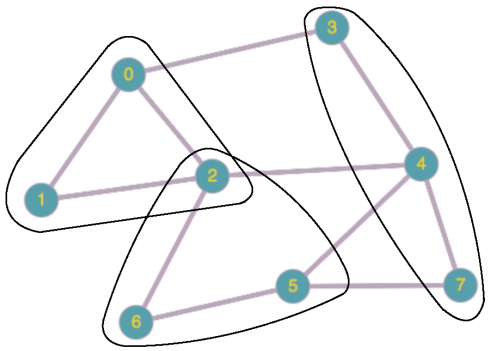


Figure 2.16: Simple example of how a WAG scheme works.

An edge is inserted into a match if both of its extreme nodes are unmatched. While it has increased time complexity (by a logarithmic factor), it provides better results than HEM and SHEM.

4. GPA (global path algorithm) [41]: it orders the edges as GEM, but builds a collection of paths and cycles of odd length. With them a matching is computed using linear programming. It has the same time complexity as GEM, but considerably better results. GPA is used in the partitioning tool KaHIP [39, 40].

Other aggregation schemes exist, such as the weighted aggregation WAG (see Figure 2.16) [42, 43]. This scheme allows a fuzzy representation of nodes, that can belong to different aggregates. This implies that at each iteration of the coarsening phase, the node set is covered by subsets that may not be disjoint. The WAG scheme propagates the connectivity of the nodes from

the original graph through all the reductions up to the coarsest level, allowing a fine yet small representation of the graph. Differently, SAG based schemes can show problems when making global and local decisions on the coarsened graphs, since some information is lost at each step. Chevalier and Safro [44] compared the two schemes in details.

Recent works modify the two schemes to improve the resulting coarsening. In [46], an efficient SAG based algorithm, MCCA, is proposed: the contraction phase is preceded by the edges weight computation, considering nodes and edges weights; then, a score is assigned to all the nodes of the graph, sorting them in descending order; the most important node is selected and forms, along with the direct neighbours, a group; lastly, the group is marked as contracted and after updating all the weights the steps are repeated, until the initial graph reaches an acceptable size. Literature also includes: tree based schemes, such as in [47], where an edge rating based on how often an edge appears in relatively balanced light cuts induced by spanning trees is introduced; a flow based coarsening algorithm [48] or even coarsening frameworks whose goal is to retain the spectral properties of a graph [49].

2.4.3.2 *Initial partition*

Once the coarsening phase reaches an end and the resulting reduced graph has a small enough size, an initial partition is defined. The algorithms that can be found in literature can be grouped into three categories: random partitioning, robust spectral partitioning [45, 14, 15] and greedy algorithms [51, 52].

- Random methods are easy to implement and have a linear time complexity, but produce very poor solutions in terms of maximum unbalance. They are usually associated to strong refinement algorithms during the uncoarsening phase.
- Exploiting eigenvectors of the adjacency matrix on a small graph, one can use iteratively spectral bisection to obtain balanced results. Implementations using the Lanczos [50] method to find the second eigenvalue of the Laplacian of the graph, allows to run the algorithm in less than quadratic time ($O(mn)$, where m is the number of iterations).
- Greedy algorithms assign one of k different random nodes (seeds) of the coarsest graph to each partition. The remaining vertices are assigned to partitions in a particular order, using greedy heuristics.

  Some of the most used heuristics are the following. GGP [51] grows a region around each seed until n/k nodes are added to the region. GGGP [51] performs the same steps as GGP but differs from it assigning vertices to a growing region minimising the total insertion gain,

that is the increment in cut size for adding free elements to a partition. The assignment is done in a circular order with respect to the partitions, and not as a whole in a single partition. MMG [52] is based on GGGP with the addition of tie breaking rules when selecting nodes to assign to a partition.

2.4.3.3 *Uncoarsening and refinement*

The last step of the multi-level partitioning approach is the uncoarsening and refinement phase. Uncoarsening is dependant on the method used for reducing the graph size. If the graph is coarsened with strict aggregation, the opposite action is easily done by unpacking the vertices in the groups and rebuilding the original structure. The situation gets more complex if instead a weighted aggregation method is used, because of the fractional nature of the coarsened nodes: when uncoarsening, there is no longer any information of the fraction of nodes assigned to higher levels. This can be resolved interpolating the node values using the values of their neighbours, to see of which aggregate they are part of.

Different refinement algorithms have been proposed over the course of the last 50 years and are still used today given their precision (with obvious enhancements). In the following paragraphs some of them are described.

KERNIGHAN-LIN (KL) LOCAL SEARCH The algorithm, proposed in [53], is an iterative, bisecting, partitioning heuristic based on swapping. For this reason it does not improve the balance of a partition, but only quality metrics associated to it. As long as the cut size keeps decreasing, the algorithm performs the following steps: node pairs belonging to different partitions which give the largest decrease or the smallest increase in cut value are exchanged and such vertices are locked and cannot be swapped again.

A cost function is used to decide which nodes to swap.

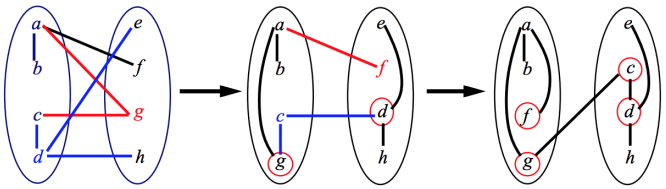
Definition 2.17. Gain of a node Given two disjoint sets A and B, the following concepts are defined.

- External cost of $a \in A$: $E_a = \sum_{b \in B} W(a, b)$, that is the sum of the edge weights crossing the set, starting from a
- Internal cost of $a \in A$: $I_a = \sum_{a' \in A} W(a, a')$, that is the sum of the edge weights inside the set, starting from a.

The difference between the external and internal costs of a node is called gain of the node.

The cost function used in the KL local search is the gain of swapping two nodes a and b (not necessarily connected by an edge), computed as:

$$g_{ab} = g_a + g_b - 2W(a, b)$$



Step #	Vertex pair	Cost reduction	Cut cost
0	-	0	5
1	{d, g}	3	2
2	{c, f}	1	1
3	{b, h}	-2	3
4	{a, e}	-2	5

Figure 2.17: Simple example of how the Kernighan-Lin local search algorithm works.

The algorithm stops when the sum of the gains associated to the swapping effectively done is less or equal than 0.

While the algorithm provides a local optimal solution (example in Figure 2.17), it has cubic time complexity and does not rebalance the partitions and is limited to the 2-way settings, requiring modifications to be applied to the general k-way graph partitioning problem. First, partitions are usually not equally sized and a solution could be to add dummy nodes to the smaller partition, only to remove them at the end of the algorithm. Second, it needs an extension to k-way, that can be done applying the algorithm on each pair of subsets. All the solutions further increase the time required by the algorithm, although the most recent and optimized versions have quadratic complexity [54].

FIDUCCIA-MATTHEYSES (FM) HEURISTIC The FM heuristic [55] improves the results of KL local search by moving single nodes between two partitions A and B instead of swapping them. The algorithm is iterative, and at each pass works as follows. FM starts by setting the state of every node to unmarked. Then, an unmarked node v with maximum gain value is selected from A and B in some way, that will be described later. The node is marked and the gain values of its neighbours are updated. This leads to two ordered sequences $a_1, \ldots, a_p$ and $b_1, \ldots, b_p$ with $a_i \in A$ and $b_i \in B$. The algorithm searches then for the smallest index such that the sum of all the node gains is maximised. If such sum is positive, the node movement is performed and the next pass begins, otherwise the algorithm stops.

The major modification with respect to KL local search is that Fiduccia and Mattheyses provide a data structure such that the computation of the node with best gain and the update of the gain values of a moved node neighbours can be done in constant time (assuming that the edge weights are non-negative integers): a priority queue (or bucket list, in Figure 2.18).

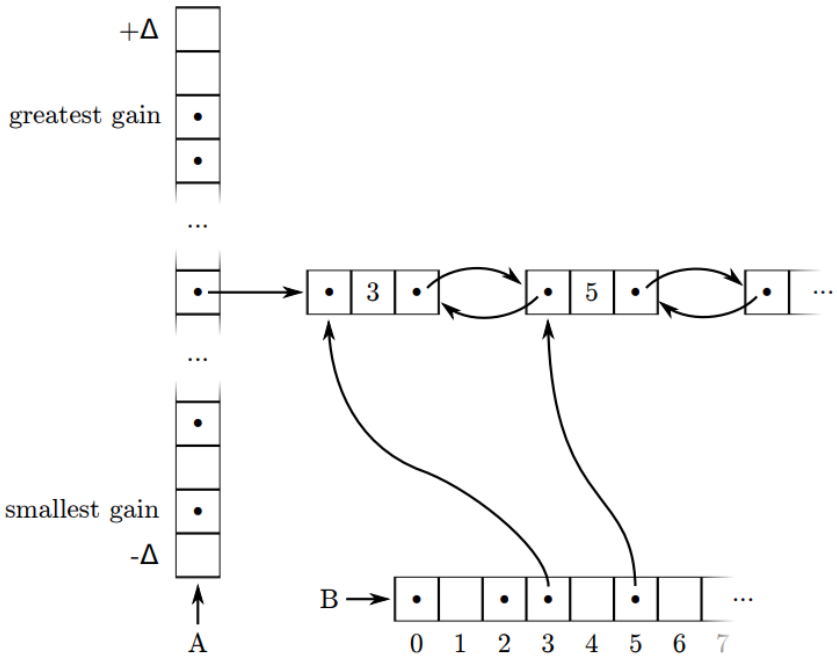


Figure 2.18: Representation of the bucket list structure used in the FM algorithm. The vector B holds pointers to nodes with the same indices, stored in some bucket of the A vector. The i^{th} bucket holds the nodes that induce a gain of i when moved to another partition.

The algorithm uses one bucket list for partition, leading to problems when allowing a certain amount of imbalance: there are two lists that can have a node moved. Holtgrewe, Sanders and Schulz [56] proposed some alternatives about how to select the candidate node to be moved.

- Alternating: the node is selected alternately from each partition.
- TopGain: the node selected is the one that provides the maximum gain among the possible movements. Ties are broken selecting a random node.
- MaxLoad: the partition with the greatest sum of weights loses a node. Ties are broken selecting a random node.
- TopGainMaxLoad: same as TopGain, but switches to MaxLoad in case of a tie.

K-WAY FIDUCCIA-MATTHEYSES (KFM) HEURISTIC An early improvement to FM heuristic was made by Sanchis, Hendrickson and Leland [57, 35]. The proposed modification uses $k(k-1)$ priority queues, one for each move (source block, target block). To move a node, all the queues maximising the

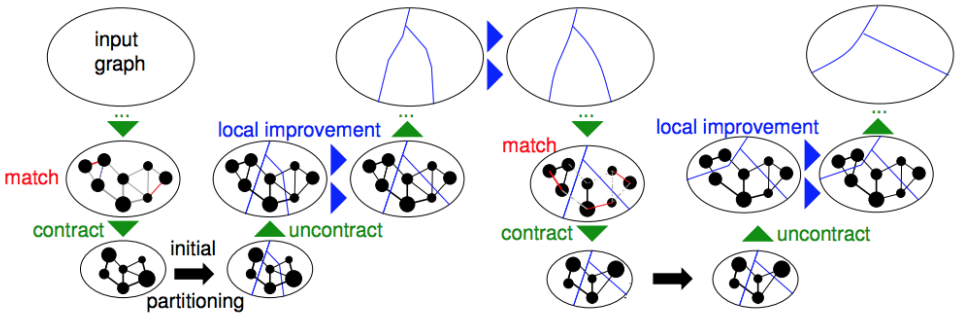


Figure 2.19: V-cycle pipeline, from [39].

gain are found; then the movement with the highest gain that improves (or maintains) the balance is performed. In other words, a node is moved to the block which maximises the reduction in the cut when the node is moved. The running time of this algorithm is, however, higher than the 2-way FM. A linear time improvement was proposed later, by Karypis and Kumar: instead of using multiple priority queues, it works with a global bucket list for all the moves.

Another improvement to FM heuristic is presented in [58]. There, instead of moving a single node for iteration, certain moves are not allowed for a specific number of iterations. The method always moves a non-excluded node with the highest gain. This algorithm is also known as Tabu based Fiduccia-Mattheyses (TFM) heuristic.

2.4.3.4 *Iterated multilevel approaches*

The main idea behind improvements of the multi-level approach is to iterate multiple times the three steps, using random different seeds for the coarsening and refinement phases, exploiting the information obtained at the end of the uncoarsening at each iteration [59].

This approach, named V-cycle (represented in Figure 2.19) works as follows: after one pass of the multilevel scheme is done, additional iterations are performed such that cut edges are not contracted; a partition can be then used as initial guess of the coarsest graph (skipping the small graph partitioning phase, done only in the first iteration). This ensures non-decreasing partition quality, given that the refinement step does not degrade the results.

In multigrid linear solvers, Full-Multigrid methods are preferable to simple V-cycles. New global search strategies were elaborated, called W-cycles and F-cycles, whose working principle differs from V-cycles in the fact that at each iteration multiple (two for F-cycles, at least three for W-cycles) re-

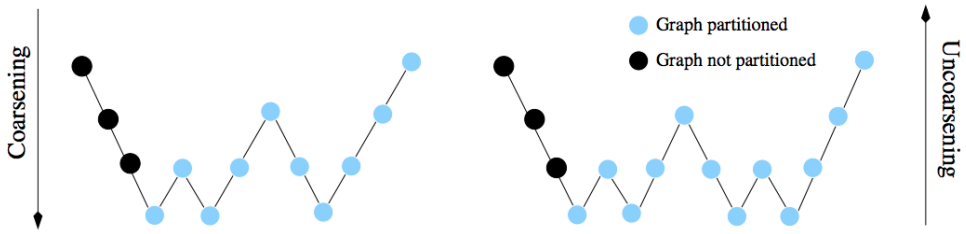


Figure 2.20: F and W cycles iteration pipeline, from [39].

cursive calls, to re-coarsen and refine an intermediate result, are performed (showed in Figure 2.20).

In the last few years, the multilevel partitioning approach has been increasingly studied, and multiple proposals have been made, all working towards a distribute approach. The majority of the state of the art algorithms present parallel characteristics: scalability is improved by performing the coarsening and refinement on different machines, as in ParMETIS [60], KaHIP [40, 61] and PT-Scotch [62].

2.5 IMPLEMENTATION OF THREE PARTITIONING ALGORITHMS

This section is dedicated to the detailed description of some of the state of the art algorithms described before 2.4. The first is a parallel implementation of the RCB geometric partitioning algorithm, included in the Zoltan software toolkit [66, 14]. The second and third described algorithms, ParMETIS [64, 65, 63] and KaFFPa [39], are both multi-level methods.

2.5.1 *Zoltan Parallel Recursive Coordinate Bisection*

The parallel RCB included in the Zoltan toolkit [66, 14, 75], starts with an uneven distribution of points over a number of k threads (remember, graphs with nodal coordinates can be represented as a set of geometric points), that are clustered into two groups. The initial assignment of the points can either be defined by the user (for example, the first processor holds $n/3$ points, the second $n/3$ of the remaining points and so on, without caring about the balance constraint) or respecting the proportion of computational power allowed by the threads.

The initial computation domain, that considers all the points, is split using a plane orthogonal to the axis corresponding to the maximum points elongation, and points are arranged according their positioning with respect to the cutting plane. For every stage, each subdomain of threads and the points that are contained in them are divided into two sets, based on which side of

the current cutting plane they belong to. Either or both of these sets may be empty.

Each thread is associated to a position in the space, determined by the cuts done. The set of points which are located in the same position in the space as the thread they are currently on, are retained on it. The remaining points are sent to the corresponding threads, on the other side of the cut. When needed, a group of threads can be further split (one common approach is to use as many threads as the number of desired partitions) and assigned to a particular region of points. To better understand this concept an example will be used, represented in Figure 2.21. Imagine a 2D case of n points (see Figure 2.21a) and 4 threads. The initial configuration is characterized by two groups, A and B containing each two threads, and the points assigned to them with a certain criterion (Figure 2.21b). The first cutting line, suppose vertical, is computed and used to partition the points (Figure 2.21c). Each group of threads is then assigned to a side of the cut (in the example group A, with threads 1 and 2, corresponds to the space to the left of the cutting line, while B to the right space). Lastly, the points are sorted parallelly among the groups, according to their positioning with respect to the separation computed (Figure 2.21c).

In order to minimize the maximum memory usage, the points that are being sent to each set of threads are distributed such that each thread in a set has about the same number of points. In the case when a processor has more points than the average number of points that the rest of the threads, belonging to the same set, have, then that thread will not receive any objects.

2.5.2 *ParMETIS k-way partitioning*

ParMETIS is an MPI-based parallel library that implements a variety of algorithms for partitioning and repartitioning unstructured graphs and for computing fill-reducing orderings of sparse matrices, showed in Figure 2.22. The procedure described here will be the one used to partition graphs [64, 65, 63].

This routine takes a graph and computes a k-way partitioning while attempting to minimize the number of edges that are cut during the execution. In the coarsening phase all the threads collaborate to compute a matching of the vertices, in parallel. More precisely, each thread computes a possible matching for the nodes in it (in particular, heavy-edge matching scheme with a balance-edge tie-breaking rule). To avoid contentious situations, such as the possible matching between points belonging to different threads, efficient communication protocols are used.

The partitioning over coarsest graph is also parallelised: threads are split into four groups, each performing a different partitioning using a task decomposition method scheme, where each processor calls the multi-constraint

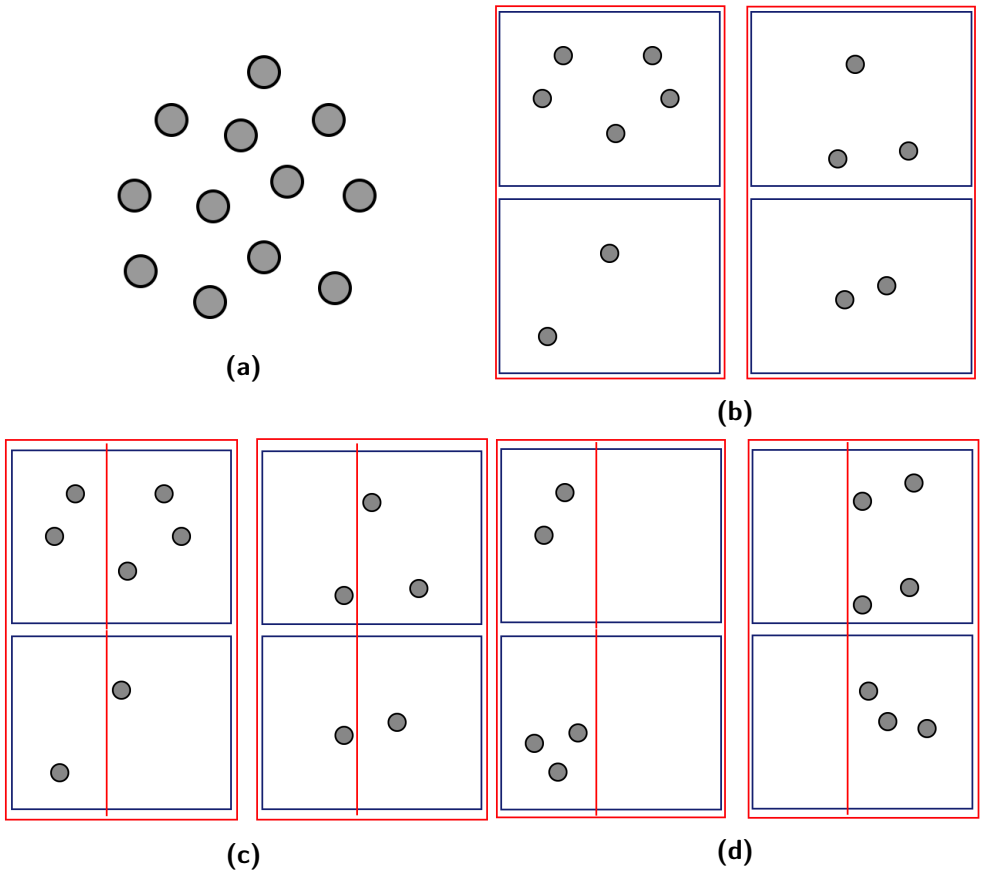


Figure 2.21: The points in Figure 2.21a are unevenly distributed to two groups of threads (groups are red boxes, threads are black boxes) in Figure 2.21b. After a cutting line is found, in Figure 2.21c, with respect to the whole points set, the nodes are reallocated to the corresponding group, in a random way, Figure 2.21d.

bisection algorithm implemented in METIS [38]. The best partitioning is then selected and communicated to all the other threads.

Lastly, the refinement phase, based on a KL/FM-type algorithm, is enhanced as a two steps heuristic. In the first step, refinement operations are made concurrently (as in the serial case), but only temporary structures are updated and used. Then, a global reduction operation is performed to check if the balance constraints are violated. If they are, each processor is required to disable a portion of its proposed node movements. An example of the refinement phase is showed in Figure 2.23.

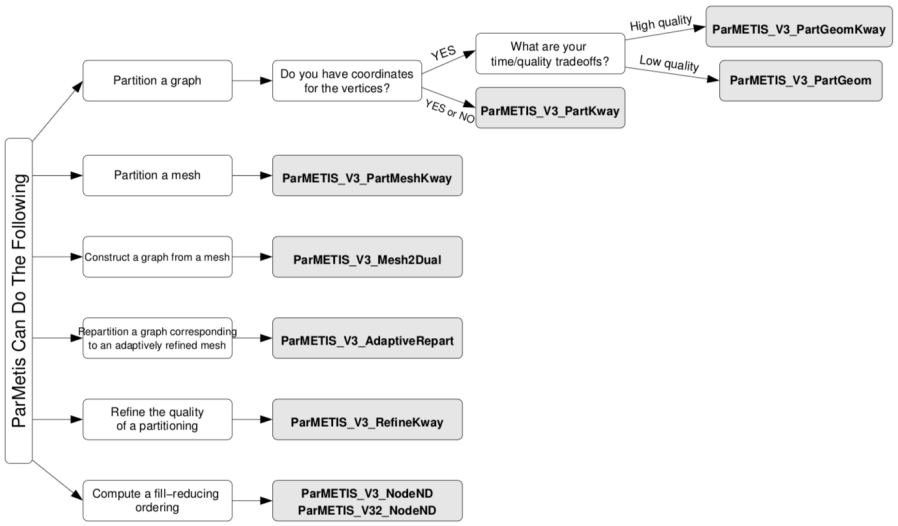


Figure 2.22: ParMETIS routines.

2.5.3 *KaHIP/KaFFPa*

Also KaHIP [40] is a collection of algorithms for partitioning and repartitioning graphs, inspired by numerous techniques (evolutionary methods, label propagation algorithms, etc.), showed in Figure 2.24.

KaFFPa [39], the algorithm that is described, is a multi-level graph partitioning framework that includes several improvements: flow-based methods, improved local search and repeated runs similar to the approaches used in multigrid solvers. As many other multilevel partitioning algorithms, the local improvement adopted in KaFFPa is a variant of the FM heuristic described before.

KaFFPa's refinement phase is organized in rounds. In each, a priority queue P is initialized with all nodes that are incident to more than one partition, in a random order. The priority is based on the gain $g(i) = max_P g_P(i)$ where the nodal gain $g_P(i)$ was defined in 2.17. A local search then repeatedly looks for the highest gain node and moves it to the partition that maximizes the total gain. In each round a node is moved at most once and after a node is moved its unchanged neighbours are inserted into the priority queue. When a stopping criterion is reached, all movements after the best-found cut, that occurred within the balance constraint, are undone. This process is repeated several times until no improvement is found.

KaFFPa additionally uses more advanced local search algorithms. The first method is based on max-flow min-cut computations between pairs of partitions. This improvement method is applied between all pairs of blocks that share a non-empty boundary (like the k-way extensions of KL and FM heuristics, it is applied to pairs of partitions). The algorithm constructs a

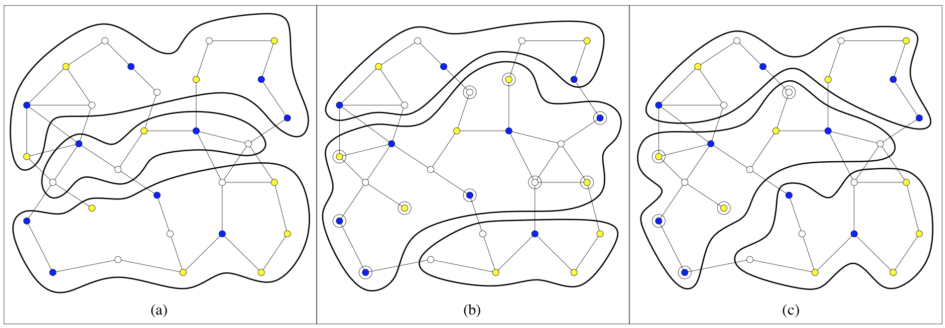


Figure 2.23: Example of the refinement scheme used in ParMETIS. In Figure (a) the represented partitions are unbalanced. Each processor (only three are considered for this case) concurrently elaborates and proposes a set of refinement moves (b). If all of them were accepted, the unbalance problem would still remain. For this reason, about half of the movements proposed are disabled, resulting in balanced partitions (c). The Figure is taken from [63].

flow problem by growing an area around the given boundary nodes of a pair of temporary partitions such that each cut in this area yields a feasible bipartition of the original graph within the balance constraint. One can then apply a max-flow min-cut algorithm to obtain a min-cut in this area and therefore a non-decreased cut between the original pair of partitions. The second method for improving a given partition is called multi-try FM. This local improvement moves nodes between partitions in order to decrease the cut. While other k-way methods are initialized with all boundary nodes, the multi-try FM algorithm is initialized with a single boundary node, thus achieving a more localized search, repeatable for several iterations. The algorithm has a higher chance to escape local optima than the KL and FM heuristics, that can get out of holes, but only to a certain extent.

It is interesting to notice that KaFFPa coarsening/refinement iteration scheme belongs the F-cycles iterative multilevel approaches described before.

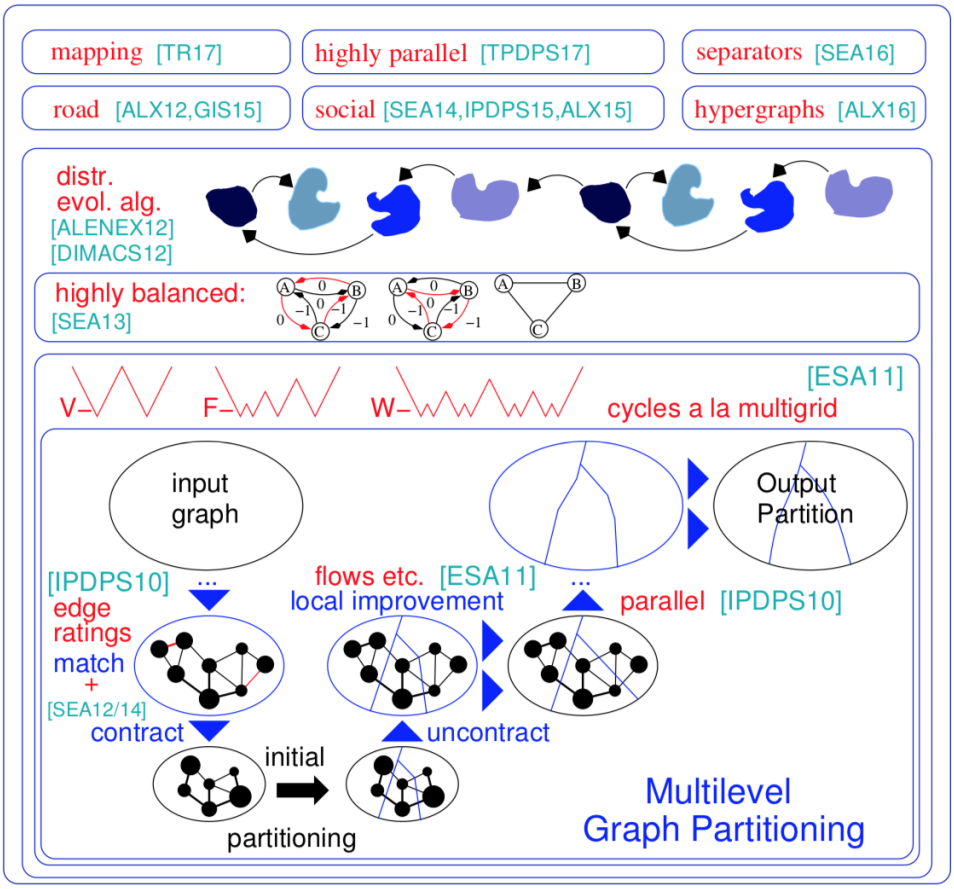


Figure 2.24: Techniques used in the KaHIP graph partitioning toolkit.

Part II

ADVANCES IN GRAPH PARTITIONING ALGORITHMS

3

LABEL CLUSTERING

The idea behind graph partitioning algorithms based on clustering is to exploit similarities between nodes, expressed as weighted edges. This similarity can be computed in different ways: number of adjacent nodes, kernels, semantic information, etc. We propose a hybrid multi-level clustering graph partitioning algorithm, where the similarity between nodes is expressed by the semantic information of the node labels. The algorithm is used to partition a graph into k components, reducing it using clustering techniques. After a brief description of the proposed algorithm, named Label Clustering, and a comparison with multi-level approaches, the chapter will contain, presented in order, the detailed steps to perform Label Clustering: graph reduction, graph coarsening and spectral clustering. Finally, experimental results to evaluate the quality of the algorithm are presented.

3.1 BRIEF ALGORITHM DESCRIPTION

Label Clustering works on a graph reducing its size clustering the nodes in two phases, then it partitions the smallest graph into k blocks, that are expanded and uncoarsened up to the original graph. The first phase consists in taking a labelled graph $G(V, E, L)$, where L is a labelling function defined over the set of nodes, and reduce it to a new labelled graph $G_R(V_R, E_R, L_R)$ that has the following properties:

- To each element $v_{R_i} \in V_R$ corresponds a subset $V_i \subseteq V$, such that $\cup_i V_i = V$ and for each distinct $v_{R_i}, v_{R_j} \in V_R$, it holds $V_i \cap V_j = \emptyset$
- Each element $v_{R_i} \in V_R$ has a label that differs from the ones of its adjacent nodes, determined by the label the nodes in the corresponding subset $V_i \subseteq V$
- The labelling function L_R has the same codomain of L

The first property guarantees that a correspondence holds between the nodes of the old and new graphs: all the nodes of G are clustered into the nodes of G_R, and all the clusters must be independent from each other. The second property is the most important, since it describes the idea behind the first reduction step: obtaining a minimized graph with respect to the node labels, such that two adjacent nodes with the same label do not exist (example in Figure 3.1). The third property grants that no label is lost in the reduction process: if in the original graph there are three labels, say red,

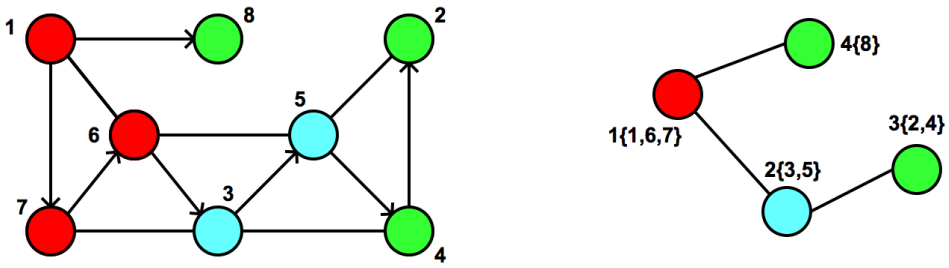


Figure 3.1: The first phase of label clustering takes the left graph as an input, that has 3 labels and 4 distinct groups of nodes, and outputs the right graph that has as many nodes as the number of groups in the input graph.

blue and green, the reduced graph will still have the same labels. The size of the reduced graph depends entirely on the initial label distribution: if the graph is labelled such that the second property mentioned before is already satisfied, the reduction is useless, since there it would be no improvement. This is an extreme case that can be considered from a theoretical point of view, but that in practice is not relevant: such graph would provide very little information with respect to an unlabelled graph, and better partitioning algorithms should be used instead.

Once the initial graph G has been reduced to G_R, depending on the obtained graph size, the second phase begins. If G_R has more than a few hundreds of nodes, a coarsening strategy is iteratively used to diminish its size. The adopted coarsening algorithm is a classic SAG max-weight reduction scheme, similar to HEM: cluster adjacent nodes with highest weights until the desired size is reached.

After a small graph is obtained, either from the reduction phase only or also with the coarsening algorithm, the low number of nodes allows a direct partitioning method. Label Clustering makes use of a version of Normalised Spectral Clustering [67], dividing the graph into k blocks, where k is a choice of the user.

3.2 SIMILARITIES WITH MULTI-LEVEL APPROACHES

Label clustering is heavily inspired by the multi-level graph partitioning algorithms described before. The three main phases are preserved, with some important differences. The fundamental one is related to the balance problem: as it is now, label clustering main goal is not to obtain balanced partitions, but to create k blocks whose nodes share the highest amount of similarity. This can be either expressed as the edge weights, representing the similarity between nodes, or some external knowledge about the label relationships (see Figure 3.2). The second difference is how coarsening is done. Instead of using a single strategy to reduce, step by step, the graph size, we

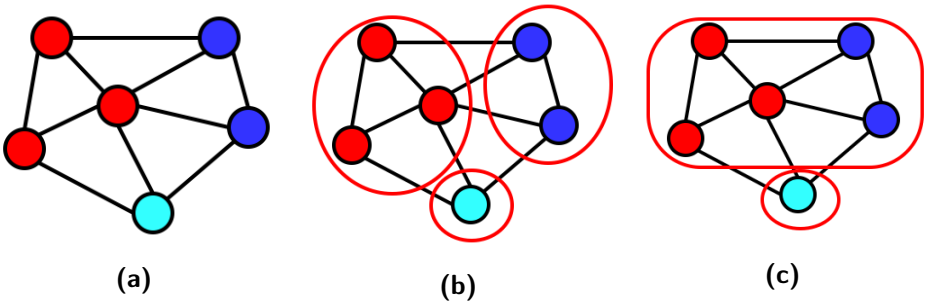


Figure 3.2: Bipartition example. Given the graph in Figure 3.2a and the knowledge that blue is more similar to red than azure, the graph is reduced aggregating nodes with the same label (Figure 3.2b) and then group aggregates according to their likelihood to be together (Figure 3.2c).

adopt two consecutive iterated phases. First, the graph is reduced in size according to the label equality, then the new smaller graph is coarsened going trough an aggregation heuristic, maximizing the similarity between nodes. The third difference is the algorithm used for partitioning the coarsest graph: Normalised Spectral Clustering [67] instead of random methods or greedy algorithms. Lastly, uncoarsening is direct with no refinement: aggregated nodes from the original graph are unpacked and associated with a partition.

3.3 GRAPH REDUCTION

Label Clustering begins grouping together all the nodes in a graph that share the same label and are connected to each other. The general concept of connection between nodes is defined as follows:

Definition 3.1. Connectivity of nodes Given an undirected graph $G(V, E)$, two distinct nodes $v_a, v_b \in V$ are connected if there exist an alternated sequence of vertices and edges $v_0, e_0, v_1, e_1, ..., v_n$ such that:

- $v_i \in V$, with $v_i \in v_0, v_1, ..., v_n$
- $e_i \in E$, with $e_i \in e_0, e_1, ..., e_n$
- $v_0 = v_a$ and $v_n = v_b$
- e_i is an edge connecting v_i and v_{i+1}

In other words, two vertices are connected if a path between them exists.

This definition of connection does not make any distinction between the paths connecting two nodes and creates ambiguous situation, as showed in Figure 3.3: a path may include only nodes with the same label or a mixture of different elements. For this reason, stricter concepts and definitions

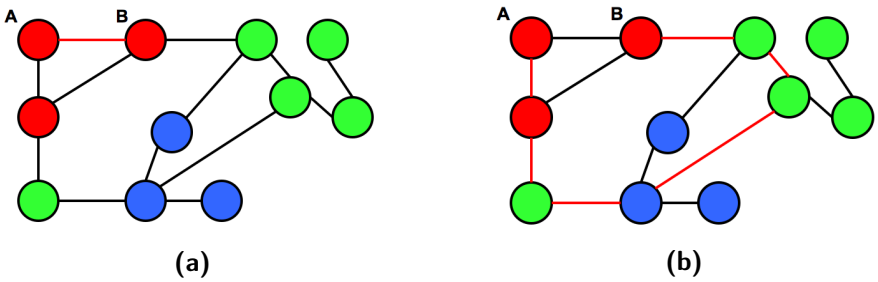


Figure 3.3: In Figure 3.3a the nodes A and B are connected directly, while in Figure 3.3b the paths between them passes through nodes with different labels.

regarding the connectivity to use when reducing a graph, are needed. First, the concept of connection between labelled nodes and labelled components are defined as follows:

Definition 3.2. Label connection Given a labelled undirected graph $G(V, E, L)$, two distinct nodes $v_a, v_b \in V$, with same label l, are *label connected* if there is at least a path P such that all of its nodes have the same label l.

Definition 3.3. Labelled component Given a labelled undirected graph $G(V, E, L)$, a labelled component $C(V_c^l, E_c^l, L^l)$ is a subgraph of G such that:

- All vertices $v_i \in V_c^l$ have the same label l
- $V_c^l \subseteq V$
- $E_c^l \subseteq E$
- L^l is a constant labelling function that assigns to each node the label l
- $\forall v_i, v_j \in V_c^l$, $i \neq j$, exists a path P between them whose vertices and edges are subsets of V_c^l and E_c^l respectively

Two important considerations can be done. First, a labelled undirected graph can always be decomposed into disjoint labelled components. Second, there is no constraint over the number of labelled components with the same label, as showed in Figure 3.4. In fact, a labelled component is determined not only by the associated label but also by its vertex and edge sets.

Two theorem can now be enunciated, one dual of the other, that will be important for the reduction algorithm explanation.

Theorem 3.1. All the nodes $v_i \in V_c^l$ of labelled component $C(V_c^l, E_c^l, L^l)$ are label connected.

Proof. The proof is direct consequence of Definitions 3.3 and 3.2. All the nodes in C have the same label, and this guarantees the first requirement

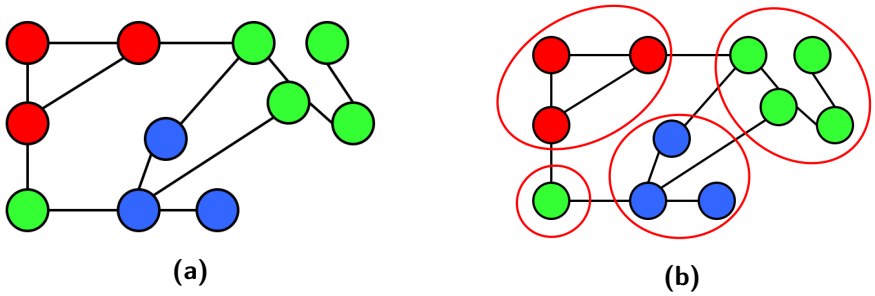


Figure 3.4: A graph with the corresponding reduction: notice how different groups have the same label.

of label connectivity. The last property of a labelled component implies that each path that can be found in it, and we can find at least one for pair of nodes, contains only nodes having the same label l, since they all belong to V_c^l. But this is exactly the second and last requirement of two nodes to be label connected. ■

Theorem 3.2. If two nodes $v_i, v_j \in V$ of graph $G(V, E, L)$ are label connected, then they belong to the same labelled component.

Proof. The proof can be derived by contradiction. Let's consider two nodes v_i, v_j that are label connected, but not in the same labelled component. Being label connected, v_i and v_j have the same label, so they potentially belong to the same labelled component (see 3.3). Since the opposite is assumed true, it must hold that there is no path between v_i and v_j such that all the nodes belonging to the path have the same label (because it is required for all the pairs of nodes in the labelled component). But this can never be satisfied because it goes against the second part of the definition of label connectivity, granted in the initial statement. ■

The task of reducing a graph using the label can be now reformulated. Given an undirected labelled graph $G(V, E, L)$ its label reduction consists into creating a new graph G_R such that:

1. The nodes of G_R are the labelled components of G
2. If two vertices of G, belonging to different labelled components, share an edge, then there exist an edge in G_R that connects the nodes corresponding to labelled components

The obtained graph is now minimal with respect to the number of nodes that share the same label and preserves the connectivity of the original graph.

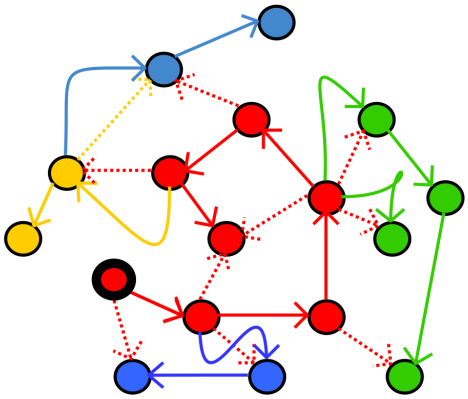


Figure 3.5: Nodes exploration, starting from the node with thick black border. Solid arrows correspond to the exploration order, dotted arrows to the visited nodes.

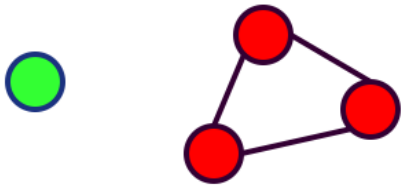


Figure 3.6: If the algorithm considered only one connected component, starting from the green node would result in a wrong output, described in the text.

3.3.1 *Basic Reduction algorithm (recursive)*

The algorithm working principle is very similar to the depth-first search on graphs and trees [70–72]. However, because the goal is to reduce a labelled graph, the algorithm must give priorities when visiting and exploring nodes, as showed in the mini-example in Figure 3.5.

The first part of the algorithm, described in the ReduceGraph procedure at line 1 of Algorithm 1, breaks the initial graph into its connected components, each used in the reduction phase. This is needed because the algorithm relies on the adjacency of the graph nodes: starting the reduction algorithm on the whole graph may lead to wrong outputs if it is not connected, violating the desired properties of minimality in terms of label connectivity. Figure 3.6 provides an example: if only one connected component was considered, the algorithm may start reducing from the green node, ending immediately because it is alone; however another component of the graph should have been reduced (the red nodes). Moreover, the reduced graph G_R is initialised with the addition of a node to indicate that a new labelled component is detected (defined as group in the algorithm).

The second procedure of Algorithm 1, ReduceConnectedComponent, is the core of the reduction algorithm. For each node the adjacency set is bisected into two disjoint sets: nodes with same label and nodes with different labels. The former are associated to the same labelled component, the current group, and are explored immediately. This *visit and explore* priority is related to Theorems 3.1 and 3.2, enunciated before. Exploring in depth-first nodes with the same label is a subtle way to search the set of nodes that satisfy the properties of a labelled component. When exploring, paths between nodes going through only sequences with the same label are enumerated, hence creating a labelled component. When the exploration of nodes with the same label is exhausted and ends, it means that a labelled component has been found and fully populated. After this, the nodes with different labels are explored: they represent initial seeds for the construction of new labelled components.

It is important to point out that in the reduction algorithm no visited node is marked. When a node searches its neighbours, it stores them distinguishing the ones with same label and the ones without it. This information is local to the node and is not propagated through the graph. While nodes with the same label are immediately explored, the ones with different labels can be seen as quarantined. Let's consider a generic node v^l, with label l. Suppose that it has adjacent nodes both with the same and different label, all not yet explored. Let also v^{dl}_{adj} be one of the neighbours with different label. Then v^{dl}_{adj} is considered visited only for v^l. If, during the exploration chain, a node v'^l explores v^{dl}_{adj}, that can happen either because they have the same label or v'^l has no neighbour with same label to explore and v^{dl}_{adj} is an adjacent node with different label, then when passing through v^l, v^{nl}_{adj} won't be explored. If no node as v'^l exist, v^l itself will explore v^{nl}_{adj}. With this explanation it can be understood how the visiting of a node is local to the nodes adjacent to it, while the exploration is global and known to all the nodes.

This property of the graph reduction algorithm is needed to avoid erroneous situations, where labelled components may result undetected or broken into multiple subsets. Such cases can still happen, depending on the graph structure and the way nodes are labelled. An example is given in Figure 3.7. Starting from node 0, nodes 1 and 3 are assigned to the same group, having equal label. Suppose that 1 is the first to be explored in the recursion: it has no neighbour with same label, but only node 2, with different label. A new group is formed and node 2 is in the same situation: its only available neighbour is 4, sharing no equality with it. Thus, the third and last group is detected. However, only two labelled components can be seen in the graph.

To solve this problem, the algorithm is applied repeatedly, until convergence. In other words, the algorithm is applied until the size of the new

Algorithm 1 Basic graph reduction (recursive)

```
1: procedure ReduceGraph(G)                          ▷ Reduction of G to G_R
2:     initialise reduced graph G_R
3:     current group cg ← 0
4:     for all connected component C_i ⊆ G do
5:         select random node v_j ∈ C_i
6:         v_j.group ← cg                              ▷ Group initialisation
7:         add node to G_R
8:         ReduceConnectedComponent(G, v, group, G_R)
9:     end for
10:    return G_R
11: end procedure

1: procedure ReduceConnectedComponent(G, v, currGroup, G_R)
2:     initialise same label nodes container SLNC
3:     initialise different label nodes container DLNC

4:     for all adjacent node v_adj of v do
5:         if v_adj.group = −1 then      ▷ v_adj does not belong to any group
6:             if v_adj.label = v.label then
7:                 put v_adj into SLNC
8:                 v_adj.group ← v.group
9:             else
10:                put v_adj into DLNC
11:            end if
12:        end if
13:    end for

14:    for all v_i ∈ SLNC do
15:        if v_i.group = −1 then
16:            ReduceConnectedComponent(G, v_i, currGroup, G_R)
17:        end if
18:    end for

19:    for all v_i ∈ DLNC do
20:        if v_i.group = −1 then
21:            currGroup ← currGroup + 1
22:            v_i.group ← currGroup
23:            add node to G_R
24:            ReduceConnectedComponent(G, v_i, currGroup, G_R)
25:        end if
26:        if ∄ edge e = (v_i.group, v.group) then
27:            add edge e = (v_i.group, v.group) to G_R
28:        end if
29:    end for
30: end procedure
```

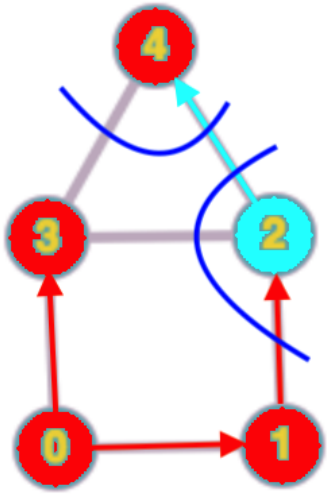


Figure 3.7: Example of how the structure of the graph and order of exploration produces more than the desired labelled components.

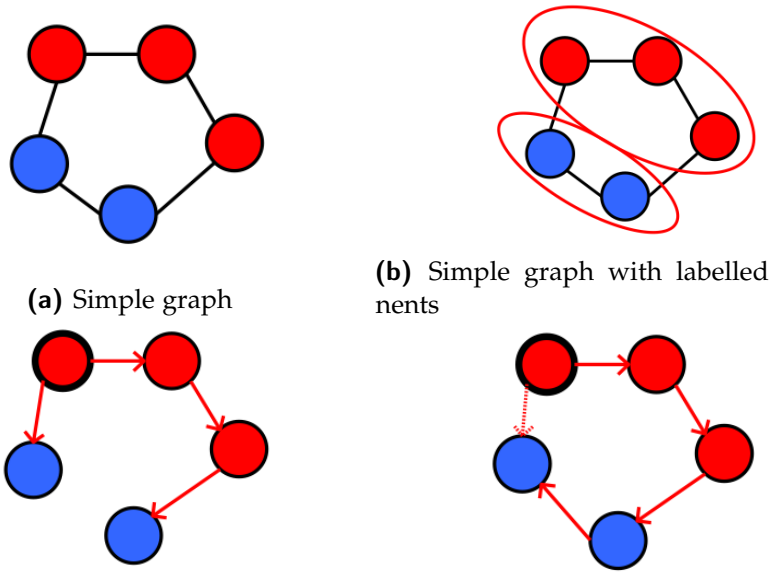
generated graph does not change, meaning that a minimal structure has been found.

Considering Figure 3.8, the small graph in Figure 3.8a can be reduced into two labelled components, as showed in Figure 3.8b. Suppose also that if a node has a neighbour that has already been visited, but not explored, by some other node, it does not visit it. This would lead to wrong reductions. As example, in Figure 3.8c the first node (marked in red) visits the blue node on the left and starts the exploration chain on the red adjacent node (having same label). When reached by the algorithm, the other blue node (rightmost) would like to explore its neighbour with the same label, but it fails, since it already visited. The obtained reduced graph has 3 nodes and it does not satisfy the minimality property mentioned in the introduction. Using instead local visiting and global exploration, as in the proposed algorithm, the desired reduction is obtained (showed in Figure 3.8d).

3.3.2 *Reduction example*

Let's now illustrate the graph reduction algorithm with an example on a simple graph. Figures 3.9 and 3.10 will be used to follow step by step the algorithm. The graph in Figure 3.9a is characterized by three labels (red, blue and green) and nine nodes. It is intuitive that the graph can be reduced into four different labelled components: this will be the main goal to pursue.

Starting from node 1, three adjacent nodes are found: 6 and 7 with the same label and 8 with different label (Figure 3.9b). Selecting the nodes mov-



(a) Simple graph

(b) Simple graph with labelled components

(c) Marking visited nodes

(d) Not marking visited nodes

Figure 3.8: Labelled components can be detected incorrectly if marking the visited nodes, as in Figure 3.8c. Our algorithm avoids marking them and produces a correct result, in Figure 3.8d.

ing counter-clockwise, the next to be explored is node 7 (Figure 3.9c). From 7, its neighbours are visited, considering that 1 is already explored: 6 with the same label and 3 with different label (Figure 3.9d). At this point, it is important to notice that nodes 1 and 7 are considered explored for every other node in the exploration set, while node 6 is considered visited for 1 and 7, that have distinct sets for visited nodes. This demonstrates the concept of local visiting and global exploration described in the previous subsection.

Continuing with the algorithm, the only candidate node for the exploration is 6, that is selected and explored (Figure 3.9e). 6 has four neighbours, but the ones with same label are already explored, so the remaining candidates for the exploration are two nodes with different label: 3 and 5 (Figure 3.9f). This means that a labelled component has been detected and populated (nodes 1,7 and 6) (Figure 3.9g).

Still going counter-clockwise, the next node to be explored is 3, being also the initial node to be part of a new potential labelled component (Figure 3.9h). Node 3 has two adjacent nodes not yet explored: 5, that has the same label, and 4, with different label (Figure 3.9i). Consequently, 5 is explored (3.9j) and its available adjacent nodes, 2 and 4, are all characterized by different labels (Figure 3.10a). Since no other node with the same label can to be explored, the second labelled cluster is detected and populated (nodes 3 and 5, Figure 3.10b).

The algorithm proceeds then in the same way as described until now: 4 is explored (Figure 3.10c) and it has only one adjacent node, 2, with same label (Figure 3.10d). 2 is then explored and no further neighbour is found (Figure 3.10e. The first depth search ends with the detection and population of the third labelled cluster (Figure 3.10f).

Now begins the backtrack through the explored nodes (in reverse order) until an element with visited neighbours that are not yet explored is found. The first and only node to have such characteristic at this point in the algorithm is node 1, whose adjacent node 8 remains isolated and unexplored (Figure 3.10g). 8 is selected but has no adjacent node. This means that 8 alone is the fourth labelled graph (Figure 3.10i). Backtracking one last time, node 1 is reached again, only to find that all the nodes have been explored. The algorithm ends (Figure 3.10j) having assigned all the nodes in the graph to a cluster.

3.3.3 *Advanced Reduction algorithm (iterative)*

As the size of the original graph increases, performing reduction recursively becomes unfeasible. Moreover, the recursive version of the algorithm presented before have, in the worst case, to be applied more than one time to reach a graph with no neighbours with the same label. We propose an iterative variant, described in Algorithm 2, that exploits a particular data structure and reduces the graph in only one pass (so that there is no need to use it more than one time). Each node is no longer associated to just its label, but to different variables that must be remembered through the iterations, even if the considered node is different. In particular, this data structure, called SnapshotIteration, contains: the node descriptor, the stack of adjacent nodes with same label, the list of adjacent nodes with different labels, the iterator of the previous list and the current stage of the snapshot.

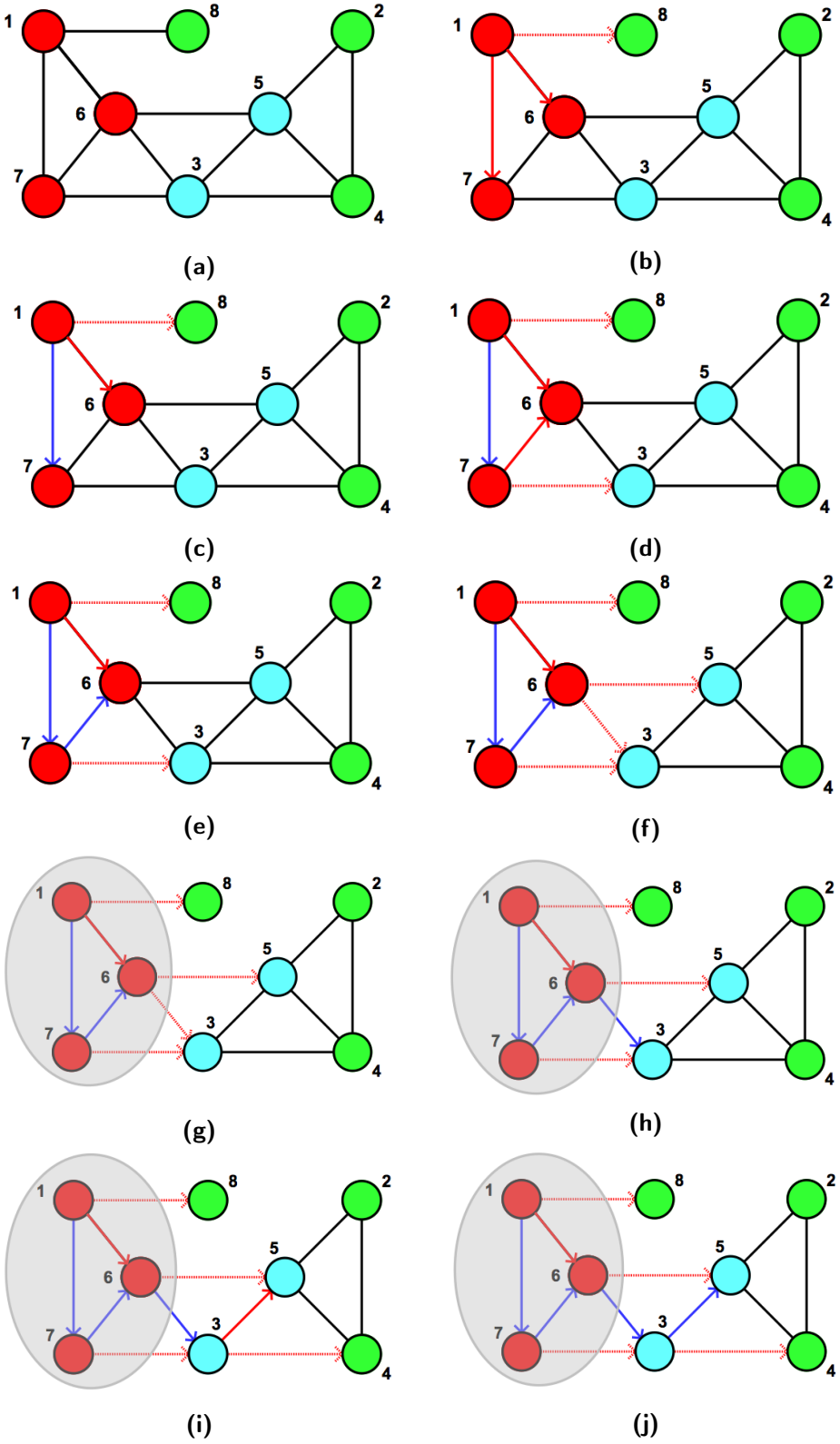


Figure 3.9: First part of the reduction algorithm example.

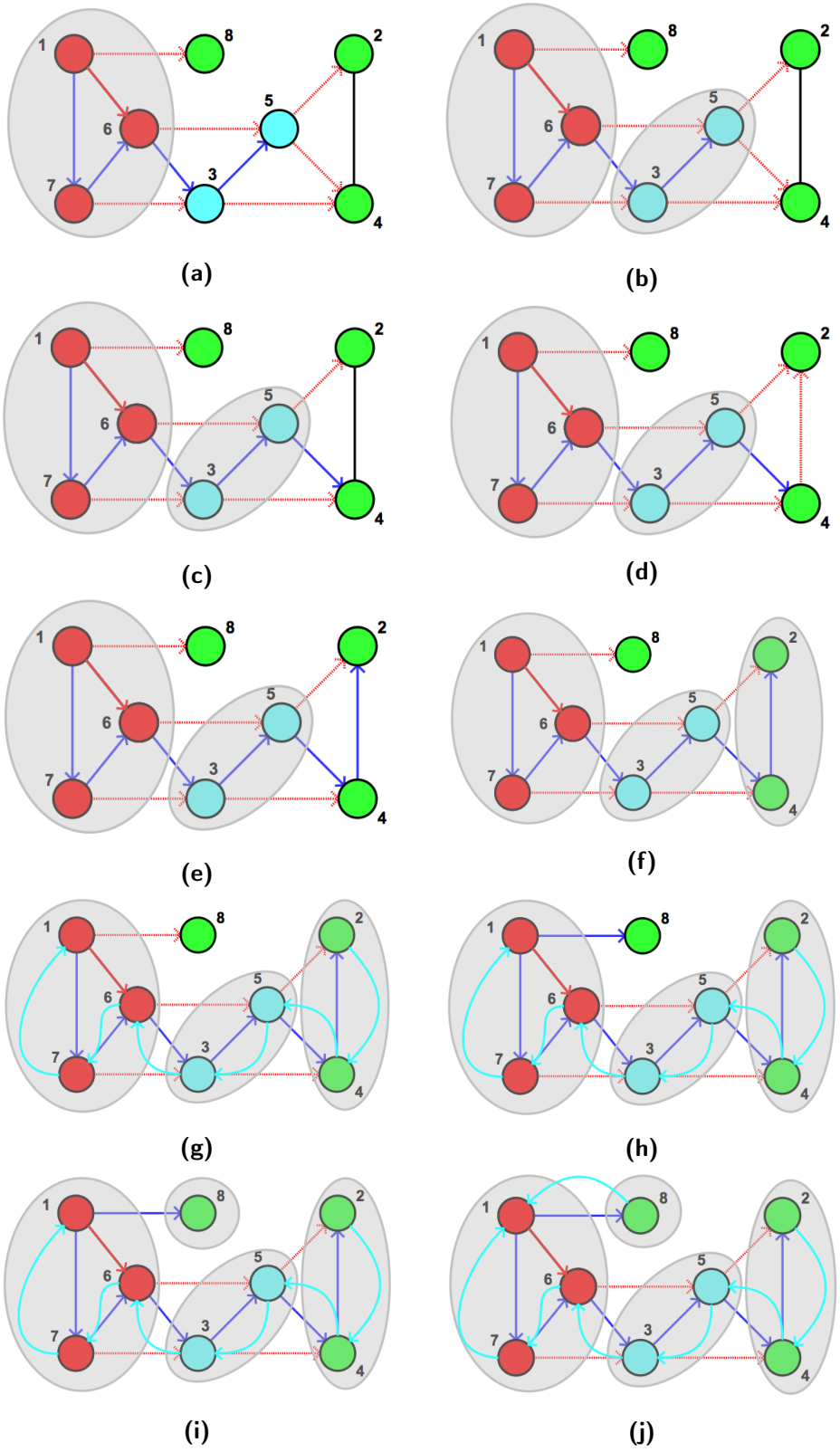


Figure 3.10: Second part of the reduction algorithm example.

Algorithm 2 Advanced graph reduction (iterative)

```
1: procedure ReduceGraph(G)                          ▷ Reduction of G to G_R
2:     initialise reduced graph G_R
3:     current group cg ← 0
4:     for all connected component C_i ⊆ G do
5:         select random node v_j ∈ C_i
6:         v_j.group ← cg                             ▷ Group initialisation
7:         add node to G_R
8:         ReduceConnectedComponentIter(G, v, cg, G_R)
9:     end for
10:    return G_R
11: end procedure

1: procedure ReduceConnectedComponentIter(G, vIn, currGroup, G_R)

2:     initialise list of SnapshotIteration structures SIL with first structure
   containing vIn

3:     while SIL is not empty do
4:         SI ← current structure considered in SIL
5:         current node v ← SI.currNode
6:         switch SI.stage do
7:             case 0
8:                 for all adjacent node v_adj of v do
9:                     if v_adj.group = −1 then
10:                        if v_adj.label = v.label then
11:                            put v_adj into SI.SLNC
12:                            v_adj.group ← v.group
13:                        else
14:                            put v_adj into SI.DLNC
15:                        end if
16:                    end if
17:                end for
18:                initialise SI.DLNC iterator SI.DLNCIT
19:                SI.stage ← 1
20:            case 1
21:                for all v_i ∈ SI.SLNC do
22:                    if v_i.group = −1 then
```

```
23:                     put new structure for $v_i$ into SIL
24:                 end if
25:             end for
26:             SI.stage ← 2
27:             if SI is not the last structure in SIL then
28:                 consider next SnapshotIteration in SIL
29:             end if

30:         case 2
31:             if SI.DLNCIT points to a valid node then
32:                 let $v_i$ be the node pointed by SI.DLNCIT
33:                 if $v_i$.group = −1 then
34:                     currGroup ← currGroup + 1
35:                     $v_i$.group ← currGroup
36:                     add node to $G_R$
37:                     put new structure for $v_i$ into SIL
38:                     consider next SnapshotIteration in SIL
39:                 end if

40:                 if ∄ edge $e = (v_i.group, v.group)$ then
41:                     add edge $e = (v_i.group, v.group)$ to $G_R$
42:                 end if
43:             else
44:                 SI.stage ← 3
45:             end if

46:         case 3
47:             if SIL has more than 1 element then
48:                 consider previous SnapshotIteration in SIL
49:             end if
50:             remove last element of SIL
51:     end while
52: end procedure
```

The only difference with Algorithm 1 is in the third phase of the main procedure, corresponding to stage 2 of a considered SnapshotIteration structure (line 20, Algorithm 2). For the first, it has been seen that nodes are visited in succession (due to the recursive nature of the algorithm), and that their order can lead to non minimal clusterings (see previous Figure 3.7). The new algorithm proposed visits and assigns to a cluster all the nodes with the same label, before creating a new group when a node with different label is encountered. This means that each labelled component is completely filled

before starting the exploration of another one, differently from Algorithm 1, where the complete filling is determined by the order of exploration (if we are lucky, nodes are visited such that a labelled component is fully considered. thing that most of the times does not happen).

The time complexity of the algorithm can be easily derived from the iterative version, with the same reasoning that are done for depth first. For each node in the graph, there are 4 iterations of the big while loop, since all the nodes pass through 4 stages. Considering an adjacency list used to represent the graph, case 0, case 1 and case 2 have at most complexity $O(2deg(i))$, if we consider node v_i ($deg(i)$ for case 0 and $deg(i)$ as sum of case 1 and case 2, since they are mutually exclusive regarding the explored nodes). Since each node will pass through all the cases cases exactly once, the complexity can be written as:

$$O(\sum_{i=1}^{|V|}(1 + deg\{i\} + deg\{i\})) = O(\sum_{i=1}^{|V|}(1) + \sum_{i=1}^{|V|}(2deg\{i\}))$$

For undirected graphs, it holds that $\sum_{i=1}^{|V|}(deg\{i\}) = 2|E|$, hence the total complexity is

$$O(|V| + 4|E|)$$

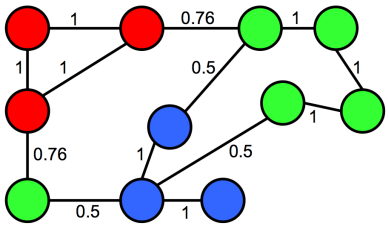
3.3.4 *True nature of the algorithm: α-reduction*

If instead of labels, their similarity values are considered, the graph reduction problem can be interpreted and described in a more complete way. In the algorithms presented earlier, the node clustering condition is having the same label. One can give a numerical interpretation to this concept. Two nodes belong to the same labelled component if they represent the same object and they are connected. If such similarity is represented with values attached to the edges of a graph, weighting them, a new structure is derived: the similarity graph. Considering a range $(0, 1]$, a small value is assigned to edges that connect nodes that are semantically distant (meaning that the corresponding labels are associated to different objects) while the weights of the edges connecting nodes that represent the same object are set to the maximum value of 1. This interpretation can be data driven (meaning that a prior knowledge of how labels are related is available), fuzzy (when a numerical value cannot be assigned to express the similarity between labels, substituted by an indicative quantity, such as low, medium or high interaction), or a mixture of the two (see Figures 3.11 and 3.12).

Although until now only nodes with the same label were aggregated, with the given interpretation this grouping constraint can be relaxed, introducing the concept of graph α-reduction. It is the same procedure described in this section, but including in the same aggregate also nodes that have a less tight

l\l	R	G	B
R	1	0.76	0.21
G	0.76	1	0.5
B	0.21	0.5	1

(a)



(b)

Figure 3.11: Considering the similarities between labels, expressed in Table 3.11a, one can assign weights to a labelled graph to express the similarity between nodes, showed in Figure 3.11b.

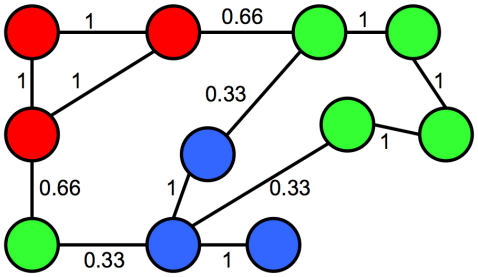


Figure 3.12: If no matrix is available to express what is the similarity between labels with a precise value, an approximation, using fuzzy values, can be imposed. For example, suppose that the similarity between labels can be Optimal(1), Good(0.66), Mediocre(0.33) or Inexistent(0). Nodes with the same label have optimal similarity, red and green share a good similarity, while green and blue have a mediocre relationship.

degree of similarity. The algorithm proposed was the strictest case: graph 1-reduction, because only the nodes with maximum similarity (same label) are grouped together.

α-reduction is extremely useful when the size of the initial graph has to be reduced quickly, retaining most of the node connectivity information. The major advantage is that, at the same cost (the algorithm remains mostly unchanged, with the exception of the weights check, that now replaces the label equality conditions), the amount of reduction that obtained from the graph can be selected. However, not all the things that shine are gold, and this relaxation brings problems.

Using α-reduction, with $\alpha < 1$, new undesired situations can happen. Consider four labels, R, G, B and O for which it is known that R and G are highly similar, as G and B are, while R and B are semantically distant and O is different from all the other labels. What may happen is that, running the algorithm, labelled components associated to different labels are aggregated together, but should not stay in the same group because of the low similarity.

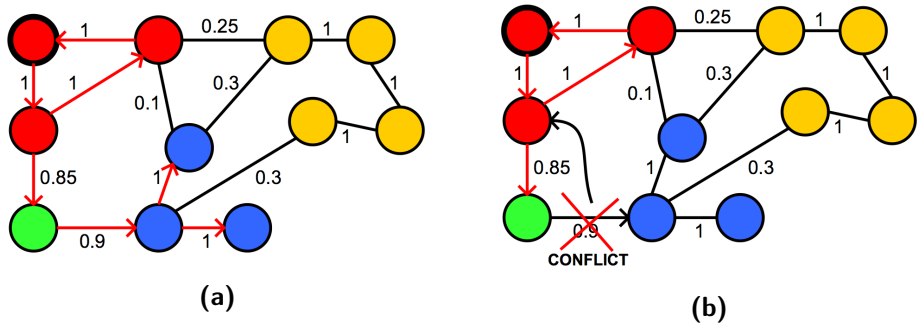


Figure 3.13: The figure represents one of the problems of the α-reduction relaxation and one way to avoid it.

An example of this is showed in Figure 3.13a: the same aggregation, obtained through the red arrows, is formed by nodes that should not belong to the same group (edge of low weight between red and blue node). A simple solution, as in Figure 3.13b, is to keep memory of the labels inserted in the current aggregation and insert a new one only if it does not conflict with them.

Another problem is related to the value of the weights associated to each aggregation. When considering 1-reduction, it is trivial to see that the groups that are formed have maximal inner weights. There is, in fact, no other way to detect labelled components with greater inner weights, since they are unique parts of a graph and the algorithm used grants maximal groups. In other words, the number of cuts to obtain the aggregations is minimal. However, the same cannot be said for α-reduction, with $\alpha < 1$. Consider Figure 3.14. There are four labels, that are all highly similar, with the exception of red-blue and red-orange. Starting from a red node, one obtains the clustering on the left, since the algorithm stops as soon as a node conflicting with one already inserted (blue conflicts with red in the example) is encountered. Such partition cuts 4 edges. Instead, if the algorithm started on a blue or orange node, the obtained clustering would be the one on the right, to which corresponds also the minimal cut.

The last problem that originates from α-reduction, with $\alpha < 1$, is how to weight the edges of the reduced graph. For 1-reduction, weights are simply the similarities between labels, since each aggregate corresponds to a single label. To solve this problem, one should consider the number of edges inside each group, the relationships between labelled components in different groups, the inner weights, etc.

Since α-reduction should be used only when the graph size is particularly big (more than a million of nodes), in the experimental section only its base form will be used on a small graph, without considering the three prob-

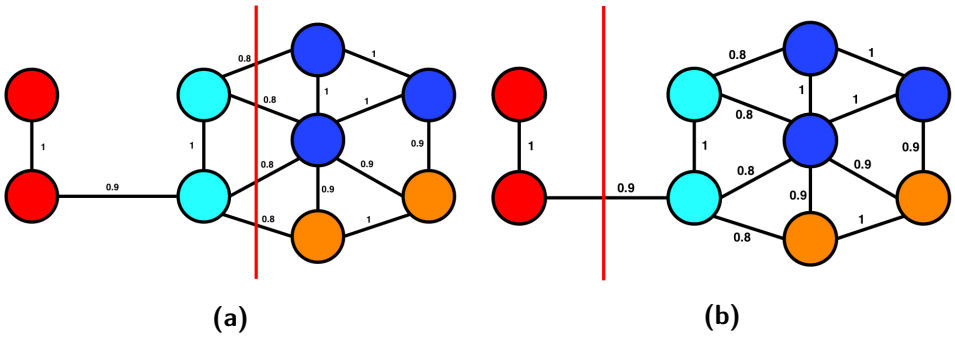


Figure 3.14: The figure represents another problem of the α-reduction relaxation and one way to avoid it.

lems and possible solutions, to provide a comparison with the 1-reduction algorithm.

3.4 GRAPH COARSENING

Once the initial graph G has been reduced to G_R, it has to be decided how to proceed. If G_R is not coarse enough, meaning that the size is still too big to apply a direct partitioning method, there is a need to further cluster the nodes. We adopt a variation of the HEM coarsening heuristic described in Chapter 2, Section 2.4.

Starting from a reduced/coarsened graph G_R, a new graph G_C is created, with smaller size (best case, half of G_R). At each iteration, the nodes of G_R are randomly selected and aggregated with the neighbour connected with the edge having highest weight. If all the neighbours of a node have already been aggregated, the node is considered as a single element group. The new nodes are connected by edges if they contain respectively two nodes that are connected in the original graph G_R. The weight of the new edge defined in the following way:

Definition 3.4. Weights of edges in the coarsened graph The edges in the coarsened graph have weights calculated as the capped product of the weights of the edges cutting the aggregates. Suppose that nodes a and b in the coarsened graph correspond to the disjoint subsets V_a and V_b in the original graph, then it holds:

$$W(a,b) = \begin{cases} \prod_{a' \in V_a, b' \in V_b} W(a,b) & \text{if this value is lower than a threshold } C \\ C & \text{otherwise} \end{cases}$$

The weight cut is needed when considering weights in range $(0,1]$, that degrade quickly as the number of iteration increases. An example of how each pass of the algorithm works is described in Figure 3.15.

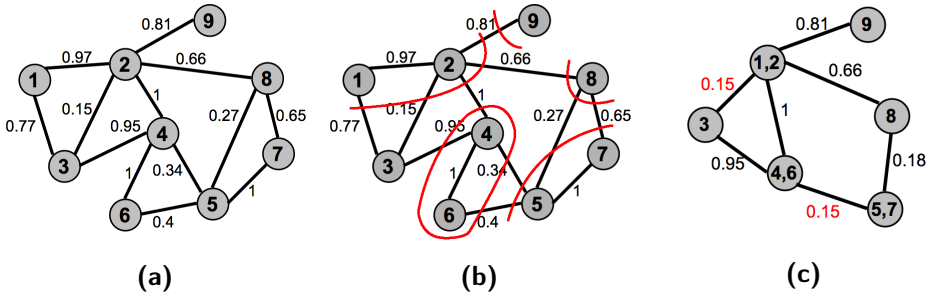


Figure 3.15: The initial graph of Figure 3.15a is coarsened (Figure 3.15b) considering the nodes in the following order: {2,9,4,7,1,3,8,5,6}. The capping rule of the example is: if the weight of an edge, calculated as product of edges crossing the cuts, is less than 0.15, bound to 0.15. Figure 3.15c shows the capped weights in red.

3.4.1 *Coarsening algorithm*

Algorithm 3 coarsens the graph until it reduces to the desired size. Each pass is composed of two steps: first, the algorithm aggregates nodes with high similarity; second, the aggregates, that are the nodes of the coarsened graph, are connected by edges.

3.5 NORMALISED SPECTRAL CLUSTERING

Once the coarsest graph is obtained, a direct partitioning algorithm can be applied on it. Instead of the methods used in the majority of multi-level graph partitioning algorithms, that are greedy or random, we use the normalised spectral clustering k-way algorithm, described in [67].

The algorithm is ill conditioned with respect of the size of the connected components in G. If there is at least one connected component formed by a single node, the corresponding rows in the similarity matrix of the graph will contain all zeros, meaning that the degree matrix will have zeroes in some spots on the diagonal. This is a problem because such matrix cannot be inverted, hence L_{sym} cannot be computed and the algorithm fails. The first solution simply puts an arbitrarily small value on the spots of the diagonals having zeros, to allow the inversion. Another more elegant solution, separates the connected components of the coarsest graph and applies spectral clustering in parallel on each. This is preferred for different reasons:

- The computational bottleneck changes from the coarsest graph size to the maximum size of the connected components of the coarsest graph, allowing to stop prematurely the coarsening phase
- Spectral clustering can be applied in parallel on the smaller sub-graphs, reducing the computational time

Algorithm 3 Graph coarsening algorithm

```
1: procedure CoarsenGraph(G, MinSize)
2:     initialise coarsened graph G_C ← G
3:     while size of G_C > MinSize do
4:         current group cg ← 0
5:         for all connected component C_i ⊆ G_C do
6:             initialize container CRNodes
7:             fill CRNodes with the shuffled nodes of C_i
8:             G_C = CoarsenConnectedComponent(G_C, CRNodes, cg)
9:             current group cg ← cg + 1
10:        end for
11:    end while
12:    return G_C
13: end procedure

1: procedure CoarsenConnectedComponent(G, CRNodes, cg)
2:     initialise coarsened graph G_C

3:     for all v_i ∈ CRNodes do
4:         initialise maxNodeValue ← 0
5:         initialise maxNode ← −1
6:         for all adjacent node v_adj of v_i do
7:             if W(v_adj, v_i) > maxNodeValue then
8:                 maxNode ← v_adj
9:                 maxNodeValue ← W(v_adj, v_i)
10:            end if
11:        end for
12:        if maxNode ≠ −1 then
13:            maxNode.group ← currGroup
14:        end if
15:        v_i.group ← currGroup
16:        currGroup ← currGroup + 1
17:        add node to G_C
18:    end for

19:    for all v_i ∈ CRNodes do
20:        for all adjacent node v_adj of v_i do
21:            if v_adj.group ≠ v_i.group then
22:                add edge to nodes v_adj.group and v_i.group of G_C if it
   does not exist
23:            end if
24:        end for
25:    end for

26:    return G_C
27: end procedure
```

Algorithm 4 Normalised spectral clustering

procedure NORMSPECTRALCLUSTERING(G,SimMatrix,k)
  using SimMatrix, build the similarity matrix of graph G, S_G
  compute the degree matrix of S_G, D
  compute the normalised Laplacian

$$L_{sym} = I_n - D^{-\frac{1}{2}} S_G D^{-\frac{1}{2}}$$

  where I_n is the identity matrix and n indicates the number of nodes in G
  compute the first k eigenvectors $v_1, v_2, \ldots, v_k$ of L_{sym}, in ascending order
  build the matrix $V \in \mathfrak{R}^{n \times k}$ containing the eigenvectors as columns
  form the matrix $U \in \mathfrak{R}^{n \times k}$ from V by normalising the rows sums to have norm 1, that is

$$u_{ij} = \frac{v_{ij}}{\sqrt{\sum_k v_{ik}^2}}$$

  consider $y_i \in \mathfrak{R}^k$, $i = 1, \ldots, n$, that is the vector corresponding to the i^{th} row of U
  cluster the points $(y_i)_{i=1,\ldots,n}$ using the k-means algorithm into clusters $C_1, \ldots, C_k$
  compute the partitions $A_1, \ldots, A_k$ such that $A_i = \{j | y_j \in C_i\}$
  return $A_1, \ldots, A_k$
end procedure

- No ill conditioning checks have to be done, since the connected components never translates into a non invertible degree matrix

3.6 LABEL CLUSTERING RESULTS

To evaluate the results of label clustering, we use images, randomly generated connected graphs and labelled meshes. In the section, one for each category is presented and briefly described.

The following configuration is used.

- Values are taken as average over different runs on a 64-bit laptop with Intel® Core(TM) i5-3337U CPU @ 1.80GHz x 4 processors, each with 3072Kb of cache size.
- The advanced reduction of Algorithm 1 is used, because we consider graph with a quite large amount of nodes.
- When performing α-reduction, $\alpha < 1$, none of the corrections described before is applied.
- If after the initial graph reduction, the obtained graph has more than 400 nodes, graph coarsening is performed.
- If a reduced graph goes through the coarsening phase, this stops when the size of the coarsened graph reaches 150 nodes.

3.6.1 *Image clustering*

Consider one of the images of KITTI benchmark suite used for instance level image segmentation [82], for which each pixel is associated to a label, in Figure 3.16.

From the image, a graph is built such that each pixel corresponds to a node, and nodes are connected if the corresponding pixels in the image are adjacent. Since the image has dimension 1242×375, the corresponding graph has 465750 nodes and 929883 edges. The reduction is done in **0.762 seconds**, and the obtained graph has only 92 nodes, corresponding to a diminishing in size of around five thousand times. The nodes corresponds to clusters of pixels having the same image, and are represented in Figure 3.17.

Since the graph is small enough, we can apply spectral clustering with inferred knowledge about the similarity between labels. Since they represent urban elements, it is trivial to associate a degree of similarity, to be interpreted as how much are two labels likely to represent objects near to each other in the real world. For example, cars should appear together with roads or parking slots, lamps have to be attached to lampposts and so on.

Figure 3.16: Image used to show the results of label clustering.

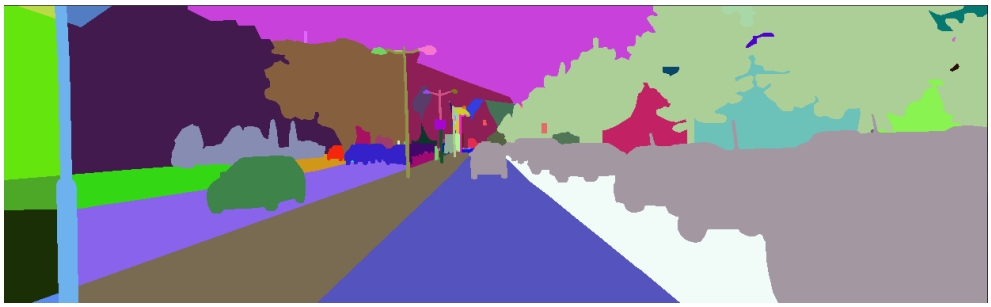


Figure 3.17: The 92 nodes obtained after one iteration of the reduction algorithm.

Figure 3.18: Image obtained back propagating clusters on the original image.

Given these considerations, the coarsest graph is partitioned into 8 groups (in **0.00831 seconds**), and uncoarsened on the original image, obtaining Figure 3.18.

Consider now a randomly generated image with five labels: red, yellow, green, cyan and blue. The image, represented in Figure 3.19 has 1166400 nodes and more than 2 million edges (size 1080×1080). Taking into account the similarity matrix showed in Table 3.1, the application of α-reduction, with different values of α, produces the following results.

- Using $\alpha = 1$, the reduction is performed in **1.882 seconds**, obtaining a new graph with 1779 nodes. This means that the original size has been

SimMatrix	R	Y	G	C	B
R	1	0.7	0.5	0.2	0
Y	0.7	1	0.3	0.1	0
G	0.5	0.3	1	0.8	0.9
C	0.2	0.1	0.8	1	0.9
B	0	0	0.9	0.9	1

Table 3.1: Label similarity matrix used to reduce the graph derived from 3.19.

reduced by a factor 656. The new graph corresponds to an image of clustered pixels, showed in Figure 3.20.

- Using $\alpha = 0.6$, the reduction is performed in **2.095 seconds**, obtaining a new graph with 400 nodes. This means that the original size has been reduced by a factor 2916. The new graph corresponds to an image of clustered pixels, showed in Figure 3.21a.
- Using $\alpha = 0.4$, the reduction is performed in **2.101 seconds**, obtaining a new graph with 181 nodes. This means that the original size has been reduced by a factor 6444. The new graph corresponds to an image of clustered pixels, showed in Figure 3.21b.

Decreasing the value of α greatly increases the reduction on the initial graph, with the cost of altered quality. With $\alpha = 1$, it is obtained a pure graph, where each node correspond to a cluster of elements in the original graph having the same label (or being connected by an edge with weight 1). With $\alpha < 1$, however, we are dirtying the reduced graph, mixing together nodes that are similar but not equal. The lower is alpha, the greater is the reduction obtained, but the worse is the mixture of elements in the new cluster.

Since with $\alpha = 1$ we obtained a graph whose size exceed the threshold fixed at the beginning of Section 3.6, we now perform graph coarsening. After several iterations, in **0.02 seconds**, the reduced graph of 1779 nodes is coarsened into a new graph of only 122 nodes. The new graph corresponds to an image of clustered pixels, showed in Figure 3.22.

It is interesting to compare the clustering obtained from the $\alpha = 1$ plus coarsening combination against the results from α-reduction with $\alpha < 1$. Thanks to the first, we obtain clusters that grows in size searching for available maximal neighbours (adjacent nodes with highest similarity), so that their representation is organized and systematic. Using the second, instead, we obtain smaller graphs in less time, but having a chaotic clustering of nodes.

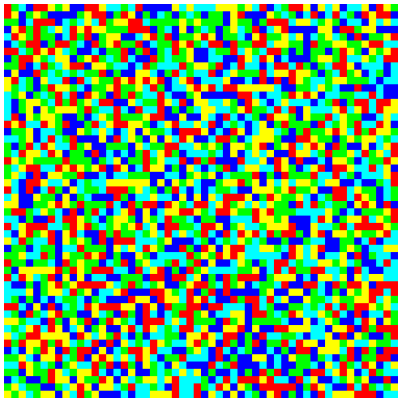


Figure 3.19: Random image with five labels.

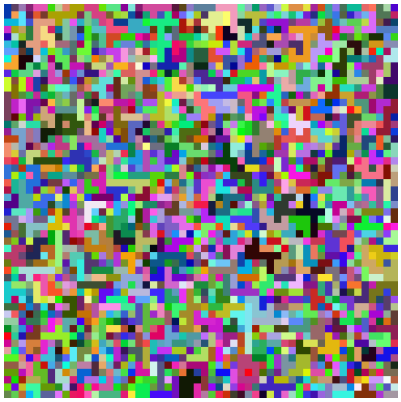
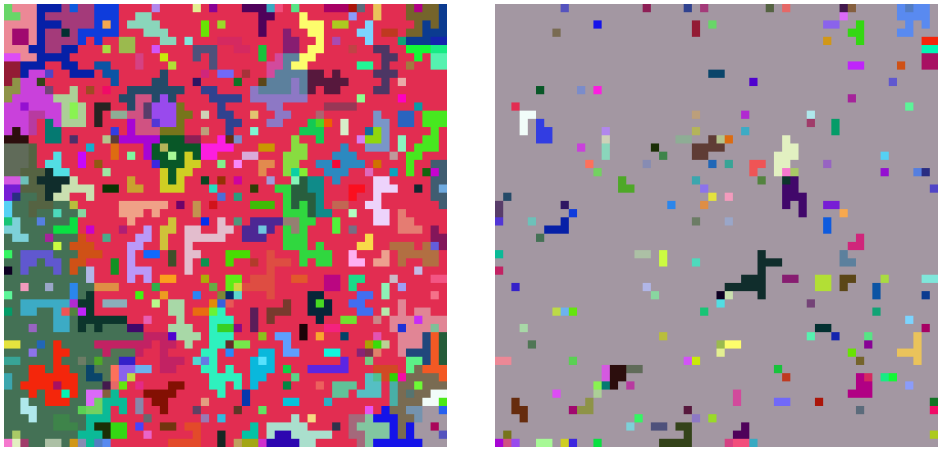


Figure 3.20: Image obtained from α-reduction, $\alpha = 1$, on the graph derived from Figure 3.19.

3.6.2 *Random graph clustering*

On random graphs, it is highly probable that more than one applications of the reduction algorithm are needed. As example, consider the chaotic graph, represented in Figure 3.23.

The graph is formed by 5000 nodes and 62870 edges, meaning that it has a discrete degree of connectivity. Moreover, 5 labels are randomly assigned to each node. In **0.0341 seconds**, the reduced graph is created, having 29 nodes (corresponding to a reduction of factor 172). The resulting graph is represented in Figure 3.24. The low number of nodes in the reduced graph is related to the fact that each node has more than ten neighbours, on average, making the graph connected and decreasing the number of labelled components.



(a) (b)

Figure 3.21: Images obtained from α-reduction, $\alpha = 0.6$ and $\alpha = 0.6$ (left to right), on the graph derived from Figure 3.19.

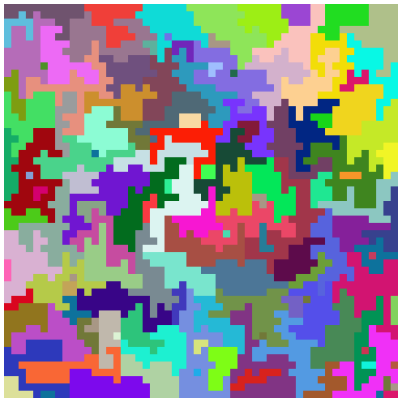


Figure 3.22: Image obtained after coarsening the graph derived from Figure 3.20.

3.6.3 *Mesh clustering*

Lastly, we show the results on two labelled meshes obtained by triangulation and refinement [78–80] of the images belonging to the EPFL [3] and South building datasets [1, 2].

The first mesh is the one derived from the fountain-P11 image sequence, in Figure 3.25. It consists in 1805233 faces associated with three labels (wall, fountain and base). From it, we obtained a new graph with 44 nodes in **4.635 seconds**, corresponding to a reduction factor 41028. The clusters can be visualised on the mesh, in Figure 3.26.

The second mesh comes from the South Building image sequence (3.27). It has 3000590 faces, associated with 3 labels (building, bushes and ground). The reduction algorithm decreases its size to a graph with only 112 nodes

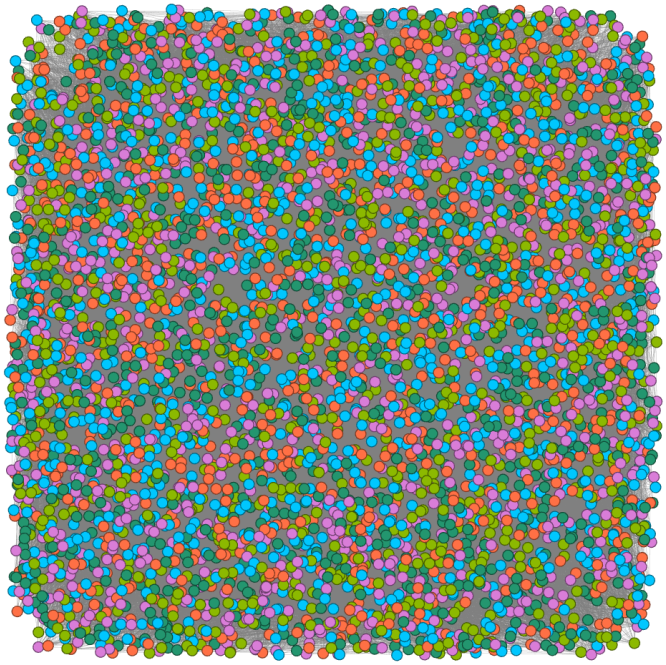


Figure 3.23: Random graph used to show the results of label clustering.

(corresponding to a factor 26791), in **7.638 seconds**. The clusters can be visualised on the mesh, in Figure 3.28.

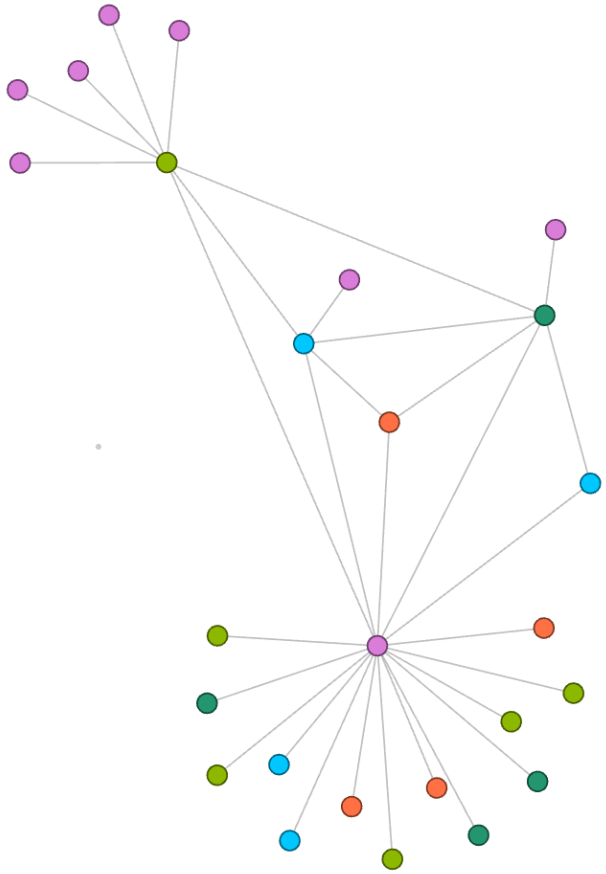


Figure 3.24: Reduced graph obtained from the graph in figure 3.23.

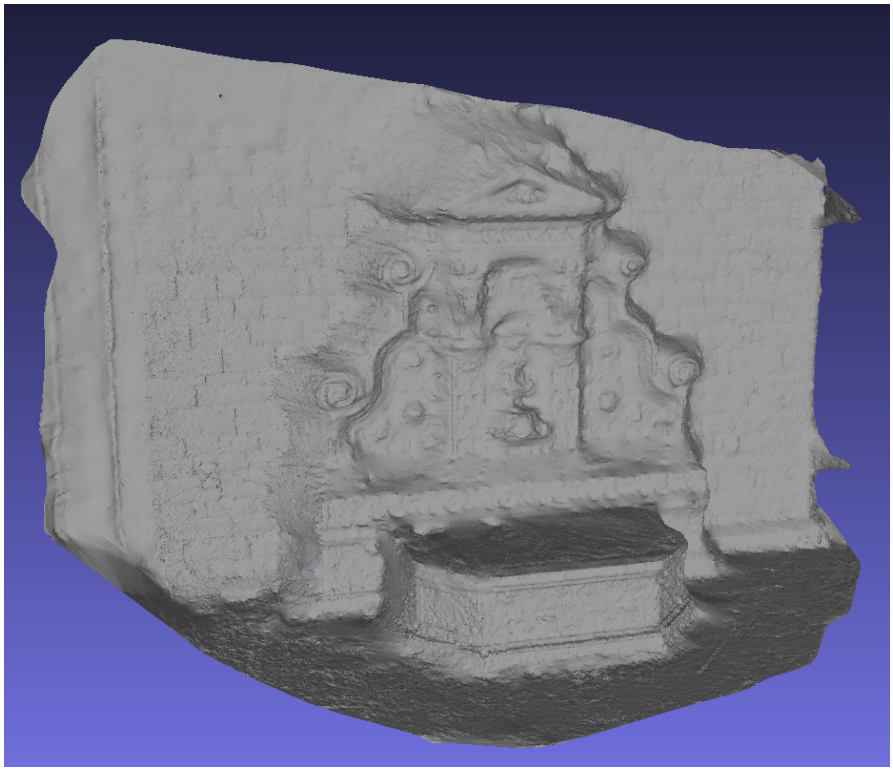


Figure 3.25: Mesh from the fountain-P11 image sequence.

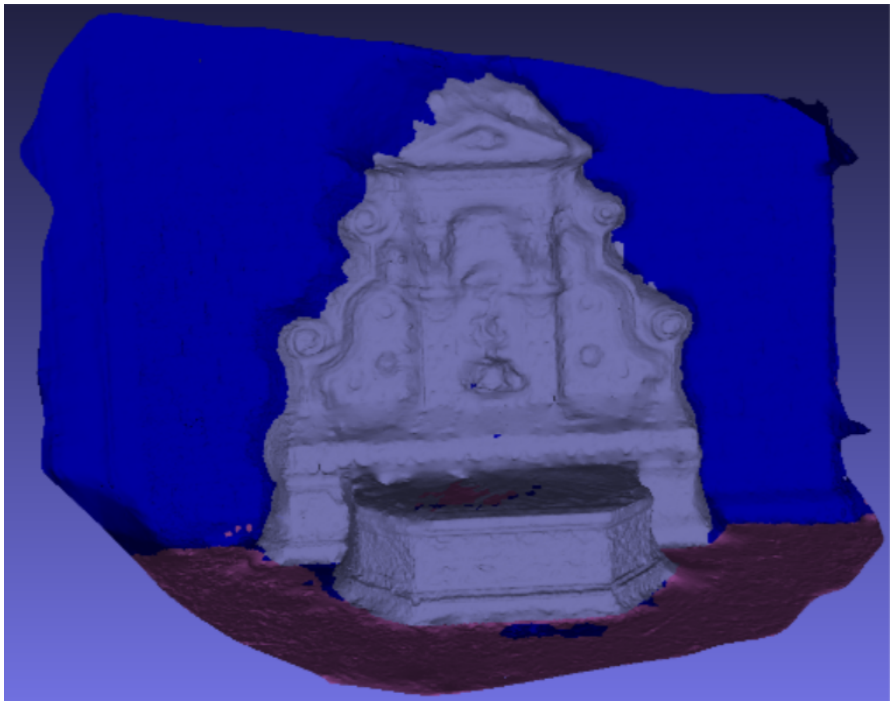


Figure 3.26: Clustered mesh (fountain-P11).

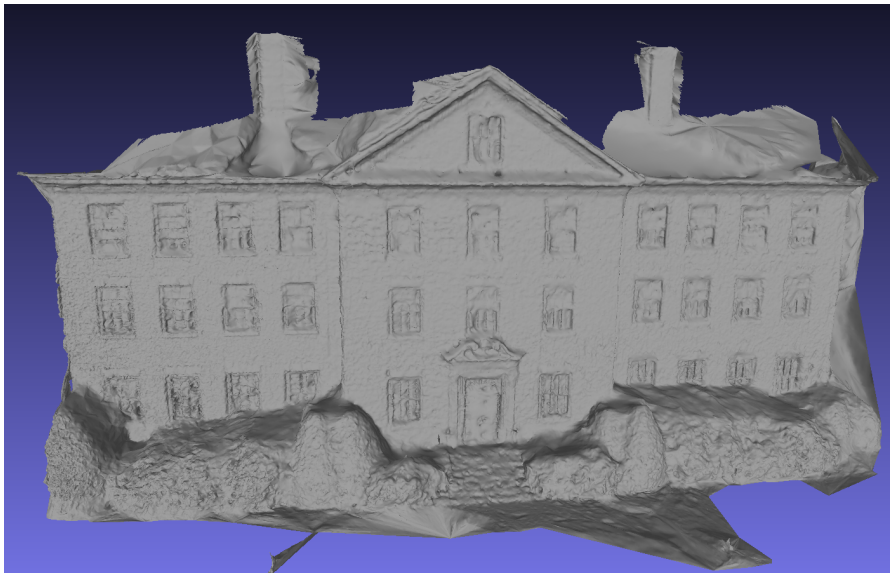


Figure 3.27: Mesh from the South building image sequence.

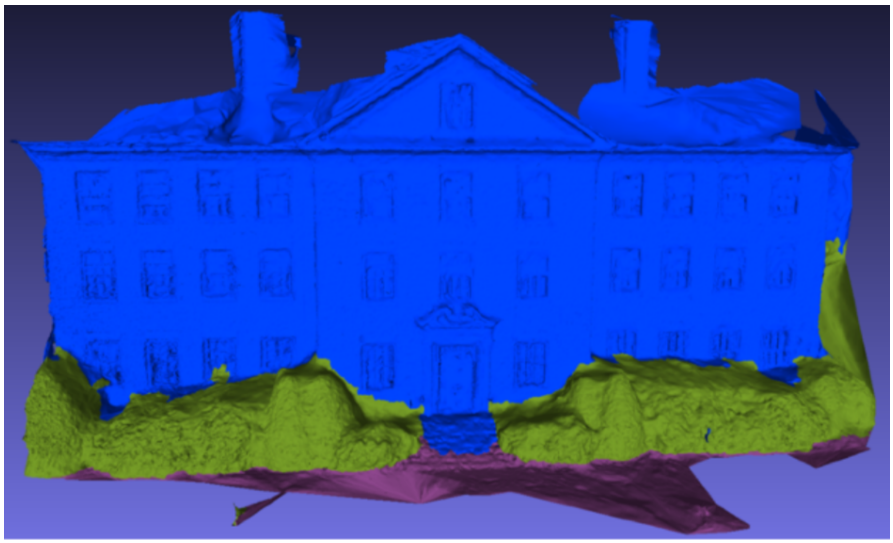


Figure 3.28: Clustered mesh (South building).

4

AD-TREE PARTITIONING

From graphs, multiple data structures can be derived, retaining all their information (node connectivity, edge weights, ...) or reducing their size. One of the latter structures is the tree, defined as:

Definition 4.1. Tree A tree T is an undirected graph in which any two nodes are connected by exactly one path.

The definition is equivalent to the following conditions:

- T is connected and with no path starting and ending in the same node
- T is connected and has as many edges as the number of nodes minus 1

Trees present different specialisations. One of them, called directed rooted tree, will be considered:

Definition 4.2. Directed rooted tree A directed rooted tree is a tree having the following properties:

1. A node is designated as root, meaning that all the other nodes generate from it
2. The edges have an assigned natural orientation
3. The level of a node is the length of the path connecting it to the root
4. The parent of a node is the node connected to it on the path to the root; every node except the root has a unique parent
5. A child of a node is a vertex that has it for parent

Moreover, the directed rooted tree can be further specialised in the following way:

Definition 4.3. Arborescence A directed rooted tree whose edges are directed away from the root is called arborescence.

An example of tree, arborescence and arborescence drawn by node levels is given in Figure 4.1. For the rest of the chapter, only arborescence are considered, and will be named trees for simplicity.

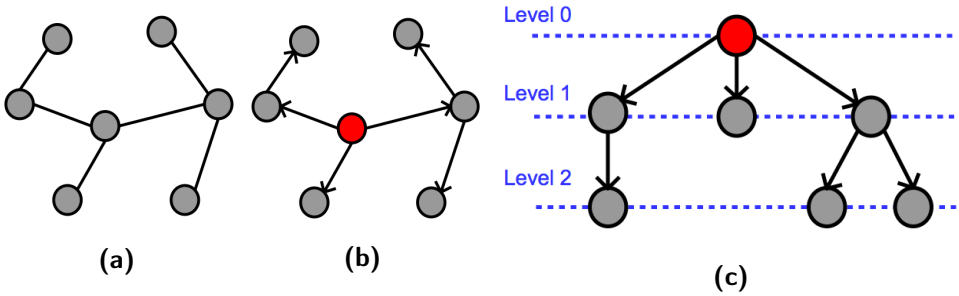


Figure 4.1: From left to right, the figures represent a generic tree, a directed rooted tree (root in red) and a well structured arborescence with levels.

4.1 PROBLEMS OF THE STANDARD TREE PARTITIONING

Using trees to partition graphs is usually avoided for different reasons. First, if undirected graphs are considered, their corresponding tree has to be built, selecting a root and spanning through all the nodes: this is an additional operation to be done aside the partitioning, worsening the time performances. Second, when building a tree from a graph some information is lost because edges and possible paths are discarded (remember, the number of edges becomes $|V| - 1$): two nodes that are connected may belong to different branches of the corresponding tree, making them oblivious of each other. Third, depending on the algorithm used to convert a graph into a tree, there may be uneven branches with different node distribution. Lastly, the balanced constraint is almost never satisfied, because of the difference of nodes contained at each level of the tree.

These problem are represented in the following figures. Consider the graph showed in Figure 4.2: it contains 10 nodes and 17 undirected edges. A possible way to build a tree from it is to select a root and visit the graph nodes using the well known breadth-first algorithm, first seen in [68, 69] (Figure 4.3). The number of edges is reduced to 9, almost half of the initial quantity. Moreover, the majority of nodes ignores the original neighbours, making them disconnected from branch to branch. As example, consider node 2: in the original graph it is adjacent to nodes 1, 4 and 7, but in the tree the node is only connected to its parent, 1. Lastly, think about partitioning the graph into five parts. It is trivial to see that all the possible cuts bring heavily unbalanced partitions: either one cuts the edge (0,7), being forced to cut all the remaining edges in the other branches, or nodes 3, 4, 6 and 8 are cut, leaving the rest as a single partition. Both results are unacceptable.

Another way to build the corresponding tree is to visit the graph nodes using depth-first [70–72], already seen in Chapter 3, after having selected a root. The number of edges is once again reduced to 9, confirming what has been said before regarding the cardinality of the tree edge set (also indepen-

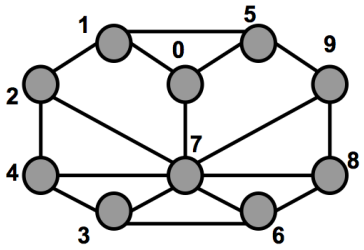


Figure 4.2: Simple graph.

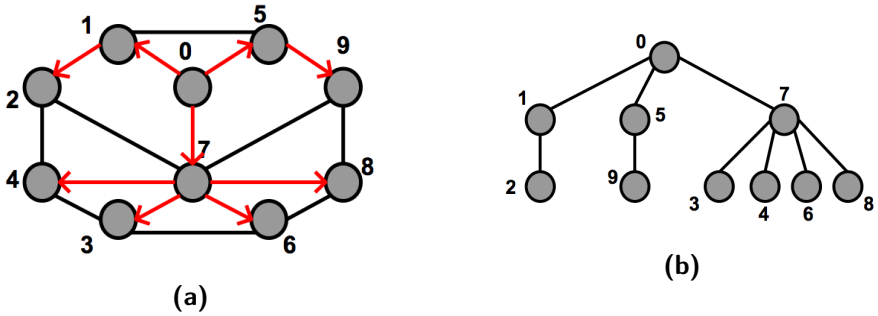


Figure 4.3: The left figure represents the tree construction on the graph using breadth-first in lexicographic order; the right figure shows the corresponding tree.

dent from the way the tree is built). Considering once again a partitioning on the tree, it is easy to notice that the balance can be satisfied, although it is not guaranteed, especially when the complexity of the graph increases. Moreover, the partitions are not minimal with respect to the cut: considering $k = 2$ in the example (Figure 4.4), perfectly balanced partitions can be obtained cutting the edge from 2 to 4, with 7 cuts in the original graph, while a better direct solution cuts only 5 edges ({0,1,2,5,9} and {3,4,6,7,8}).

To mitigate these problems, we propose a new data structure to represent a graph as a tree, retaining most of the information regarding the connectivity between nodes. This data structure is used in a new partitioning algorithm based on the propagation of subtrees size and tree traversal, whose purpose is to achieve fast results, independently from the number of desired partitions, and a good balancing.

4.2 ENHANCED TREE DATA STRUCTURE: AD-TREE

As seen before in Section 4.1, considering a naive tree construction from a graph is not sufficient to obtain good partitions, because the majority of the information related to the graph connectivity is lost. We present a new data structure, the AD-tree, used to support the proposed tree based partitioning algorithm. To improve a normal tree two considerations are made. First,

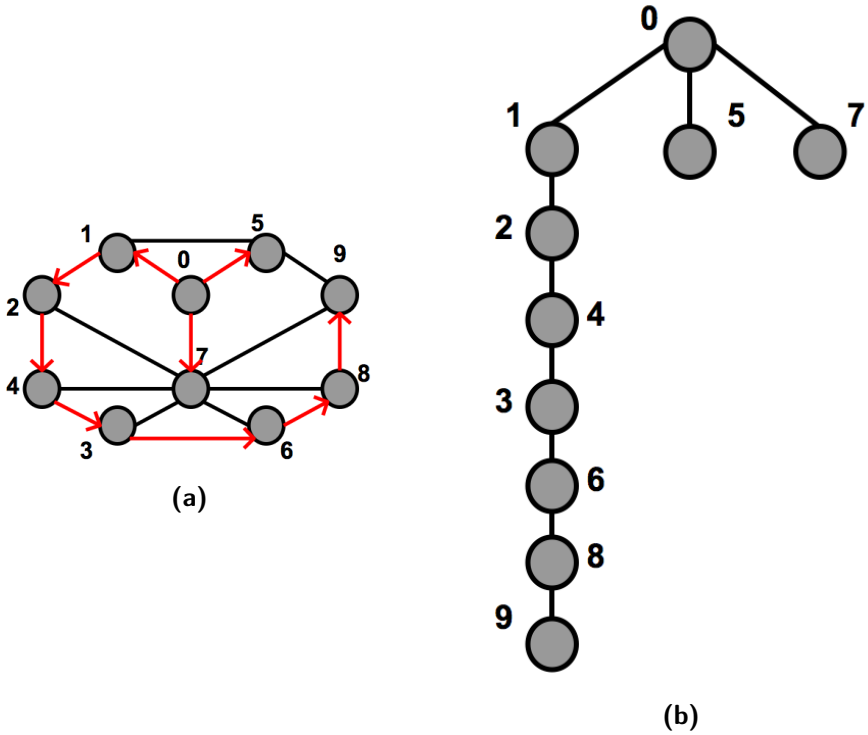


Figure 4.4: The left figure represents the tree construction on the graph using depth-first in lexicographic order; the right figure shows the corresponding tree.

every edge of the original graph should be considered, retaining the whole connectivity of it. Second, a tree should be traversed in a fast way, allowing the nodes to have not only parent and children, but also connections to other nodes, respecting the order of insertion in the tree. These improvements are named descendants graph and Ariadne's tree, and are described in the two next following subsections.

4.2.1 *Descendants directed graph*

The first enhancement that we propose on the standard tree data structure is relative to the connectivity. When building it the tree, instead of cutting edges when encountering already inserted nodes, a pseudo-link is inserted between the nodes.

Given an undirected graph $G(V, E)$, the corresponding tree T_G is built with a visiting algorithm, like breadth-first or depth-first. Focusing on one step of the construction, let $v \in V$ be the current node that is explored. One of its adjacent nodes is its parent (except the case where v is root, since it has no parent), and should not be considered. Suppose that another neighbour, u, is already part of the tree built up to that moment, before v is reached, in a

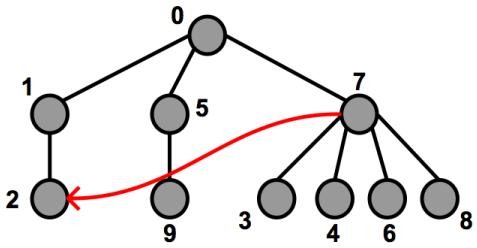


Figure 4.5: Descendant example.

different branch. Instead of not considering the edge connecting them, as in the normal tree building procedure, a directed pseudo-link is created, such that:

- u points to v if the level of u is lesser than v
- v points to u if the level of u is greater than v

The pointed node is called *descendant* of the pointing node. For duality reasons, the pointing node is called *ancestor* of the pointed node.

Consider again the graph in Figure 4.3a. When reaching node 2, one of its neighbours is node 7, but it already belongs to the tree. Since 7 has lower level (closer to the root) than 2, we insert a pseudo directed edge from node 7 to node 2; 2 is considered a descendant of 7, and 7 is an ancestor of 2 (showed in Figure 4.5).

Different rules are needed regarding how to insert these pseudo-edges:

1. A pseudo-edge connects two nodes in the tree if they are not one parent of each other and in the original graph they are connected by an edge
2. A pseudo-edge is always directed from the node with lowest level to the one with highest level
3. If two nodes, connected by a pseudo-edge, have the same level, the edge is directed towards the node that is explored last

Referring again to the graph in Figure 4.2, the fully enhanced tree with pseudo-edges is represented in 4.6.

Considering only the nodes and the pseudo-edges, a new structure is obtained: the descendants graph. It is a directed graph with multiple connected components. At least one of these is formed by a single node: the root, that explores all the adjacent nodes and is never associated with descendants or ancestors. Another interesting property is that the degree of each node in the enhanced tree is the same as the degree of the same node in the graph: this is important because we retain the knowledge of all the edges in the graph, increasing the number of total edges of the tree from $|V-1|$ to $|E|$.

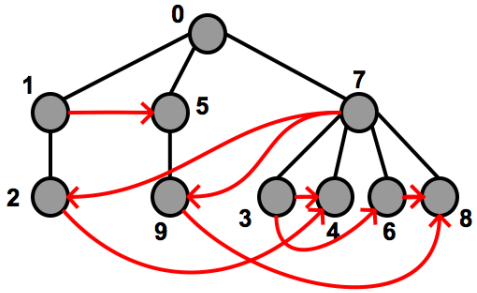


Figure 4.6: Tree enhanced with pseudo-edges indicating the descent of the nodes.

The following theorem demonstrates that a pseudo-edge cannot appear across nodes belonging to the same branch.

Theorem 4.1. Given two nodes $u, v \in V$ such that u belongs to the path connecting v to the root, no pseudo-edge can connect u and v, in any direction.

Proof. We prove the theorem by contradiction. Consider two nodes $u, v \in V$, such that u belongs to the path connecting v to the root and there is a pseudo-edge between them. In particular, following the rules stated before, the pseudo-edge is known to be directed from u to v, because u has lower level. The fact that a pseudo-edge exists implies that u and v are adjacent nodes in the original graph. This situation is possible only if u is the parent of v, because otherwise the construction of the tree would be faulty: u visits all of its neighbours that are not already in the tree and that are not its parent. v is neither its parent, because u belongs to the path connecting v to the root, nor it is already present in the tree for the same reason, so it must be one of the children of u. But this violates the rule stated before that parent and child cannot be connected by a pseudo-edge. ■

4.2.2 *Ariadne's tree*

Everyone knows the story of princess Ariadne, in Greek mythology. King Minos attacked Athens after his son was killed there. The Athenians asked for terms, and were required to sacrifice seven young men and seven maidens to the Minotaur inside a labyrinth, which construction was ordered by Minos itself, every seven or nine years. One year, the sacrificial party included prince Theseus who volunteered to come and kill the Minotaur. Ariadne, that was put in charge of the labyrinth by her father Minos, fell in love at first sight, and helped him by giving him a sword and a ball of thread so that he could find his way out of the Minotaur's labyrinth.

In the same way, the tree is enhanced with a thread that connects every node, in order of expansion, starting from the root. This chain is itself a directed tree, having a root (the same of the normal tree), and $|V| - 1$ directed

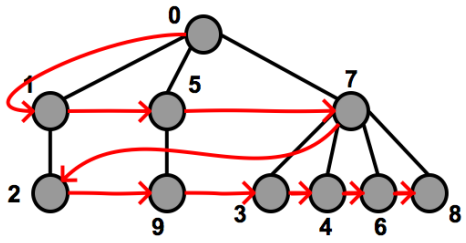


Figure 4.7: Normal tree enhanced with the Ariadne's tree.

edges, hence the name Ariadne's tree. Ariadne's tree is important when partitioning the normal tree, because it allows to quickly traverse it from the last node explored to the root. Its usefulness can be understood better when explaining the partitioning algorithm working principle. Considering one more time the graph in Figure 4.2, with the corresponding tree, showed in Figure 4.3b, its Ariadne's tree is represented in Figure 4.7.

The most important property of the the new data structure is that, given a graph and an algorithm to build a tree from it, there is one and only one Ariadne's tree that can be built on it. Obviously, changing the algorithm used to derive the tree from a graph also changes the corresponding Ariadne's tree.

4.2.3 *AD-tree and its properties*

Summing up all the improvements, we created a new data structure to represent a graph, the AD-tree (Ariadne/Descendants tree), formed by three components: the standard tree, the Ariadne's tree and the descendants graph. To build the AD-tree the breadth-first visit of the graph is used, starting from a random node. If the original graph has multiple connected components as many AD-trees as their number are obtained.

Before describing the algorithm used to create the AD-tree, few words should be spent to describe the single node of the new structure. Consider Figure 4.8. A node of the AD-tree is a composed by the following elements:

- A reference to the node parent (node->parent)
- A reference to each children (node->children[i])
- A reference to each descendant of the node (node->descendants[i])
- A reference to each ancestor of the node (node->ancestors[i])
- A reference to the next node in the Ariadne's tree (node->next)
- A reference to the previous node in the Ariadne's tree (node->prev)

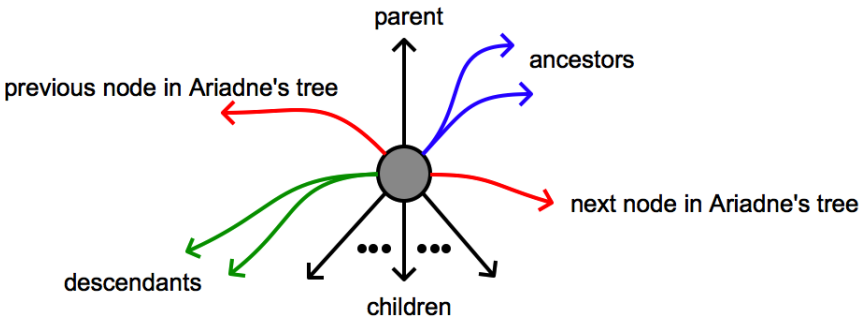


Figure 4.8: Single node of the AD-tree.

- An identifier of the node

As for every other type of tree, the root does not have a parent and the leaves do not have any child. Moreover, the root does not have descendants and ancestors, because by construction it visits immediately all the adjacent nodes. The root of the AD-tree is the same root of both Ariadne's tree and the normal tree derived from the graph. Figure 4.9 shows, for graph in 4.9a, the corresponding tree, descendants graph and Ariadne's tree, including all three together to form the AD-tree.

Considering a graph $G(V, E)$ with a single connected component, the corresponding AD-tree ADT manifests the following properties, that can be easily extended for graphs with multiple connected components:

1. ADT has the same nodes of G
2. ADT has $|E| + |V| - 1$ edges, both directed and undirected
3. For every node of ADT, the sum of the number of children, descendants, ancestors and the parent is equal to the degree of the same node in G
4. The level of a descendant is always greater or equal than the level of the corresponding ancestors, because the descendant is farthest from the root
5. The level of an ancestor is always lesser or equal than the level of the corresponding descendants, because the ancestor is closest to the root
6. The descendant of a node cannot be also the node ancestor: the first node that is explored during the creation is considered the ancestor

The first property is trivial, since creating a tree from a graph preserves the number and identity of the nodes. The second property can be derived by the combination of the AD-tree components. The naive tree is a directed

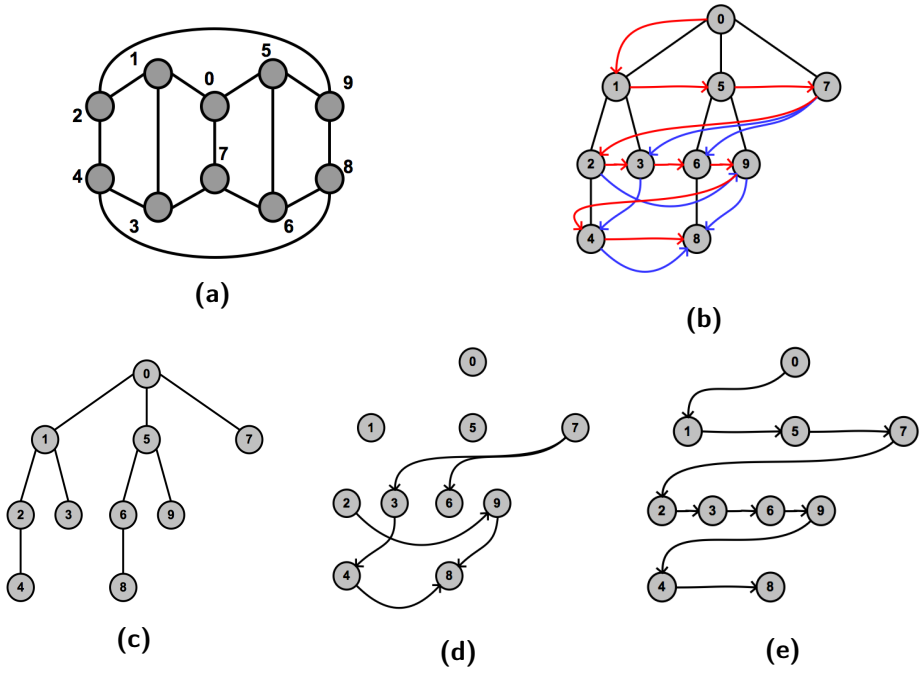


Figure 4.9: From left to right. In the top row, a graph and the corresponding AD-tree. In the bottom row, the AD-tree is broken into its components: the standard tree, the descendants graph and Ariadne's tree.

rooted tree, that has $|V| - 1$ edges; the descendants graph is formed by the edges that are not part of such tree, so it has $|E| - (|V| - 1)$ connections; Ariadne's tree is also a tree, hence having $|V| - 1$ edges. Summing all the three values together, since the AD-tree is obtained as superimposition of the three components, $|E| + |V| - 1$ edges are obtained.

The third property reflects the fact that all the edges outgoing a node in a graph are reported either in the naive tree or in the descendants graph, that by definition is the set of edges of the original graph that are not inserted into the normal tree. The fourth and fifth property have already been described when introducing the descendants graph.

The sixth and last property, implies that there cannot be a cycle of descendant elements, that may happen if they have the same level. A convention is adopted to solve ambiguous situations: the node visited first during the AD-tree construction algorithm is considered the ancestor. In other words, if two nodes u and v have the same label and belong to the descendants' graph, if u belongs to the path starting from the root and ending in v, on Ariadne's tree, then u is an ancestor of v, and v is one of its descendants.

Algorithm 5 Construction of the AD-tree(s) of a graph

```
1: procedure ADTreesCreation(G)
2:     initialise container of roots ADRoots

3:     for all connected component C_i ⊆ G do
4:         select random node v_j ∈ C_i
5:         ADTreeCreation(G,v_j)
6:         insert v_j into ADRoots
7:     end for

8:     return ADRoots
9: end procedure
```

4.3 AD-TREE CONSTRUCTION

Now that the new structure has been defined and described, this section will explain the algorithm used for its construction, including possible variants and improvements.

4.3.1 *Construction algorithm and time complexity*

The algorithm used to create the AD-trees from a graph first breaks it into it connected components (Algorithm 5). For each component, it builds simultaneously all the components of the corresponding AD-tree (Algorithm 6). The working principle is the same as breadth-first: simulating a queue to store the nodes of the graph/tree, it takes the front element and searches the neighbours not visited, making them children of the front node (and conversely, the front node is made their parent).

The difference from breadth first is that a cursor is used to build both the normal tree and Ariadne's tree while the nodes are visited and explored: when a new node is inserted into the tree, the last element of Ariadne's tree is chained to such node, that becomes the new tail (lines 14, 15 and 16 of Algorithm 6): this is the same as a queue. In other words, nodes are inserted into Ariadne's tree according to their visitation order, using the structure as the queue used in breadth-first.

If a neighbour is already inserted, it means that it is either a descendant or an ancestor, depending on its level. Particular attention should be given to the special case of level equality: as already stated, the convention used is that if two nodes have the same level and have the ancestry-descent requisites (belonging to different branches, not children of each other and connected in the original graph), the one that is explored first is the ancestor. The algorithm works checking if this convention is followed. Suppose that

Algorithm 6

```
1: procedure ADTreeCreation(G,vInit)
2:     initialise Ariadne's tree cursor atc ← vInit
3:     initialise Ariadne's tree tail cursor attc ← vInit
4:     initialise flags (to false) to see if a node is already in the tree or not isNodeInTree
5:     isNodeInTree[vInit] ← true
6:     vInit.level ← 0

7:     while atc points to an existing node do
8:         current node v ← atc
9:         for all adjacent node v_adj of v do
10:            if isNodeInTree[v_adj] is false then
11:                insert v_adj into v.children
12:                v_adj.parent ← v
13:                v_adj.level ← v_adj.parent.level + 1
14:                v_adj.prev ← attc
15:                attc.next ← v_adj
16:                attc ← v_adj
17:                isNodeInTree[v_adj] ← true
18:            else
19:                if v.level < v_adj.level then
20:                    insert v into v_adj.ancestors
21:                    insert v_adj into v.descendants
22:                else
23:                    if v.level > v_adj.level and v.parent is not v_adj then
24:                        insert v into v_adj.descendants
25:                        insert v_adj into v.ancestors
26:                    else
27:                        if v.level = v_adj.level then and v_adj ∉ v.descendants and v_adj ∉ v.ancestors
28:                            insert v into v_adj.ancestors
29:                            insert v_adj into v.descendants
30:                        end if
31:                    end if
32:                end if
33:            end if
34:        end for

35:        atc ← atc.next
36:    end while
37: end procedure
```

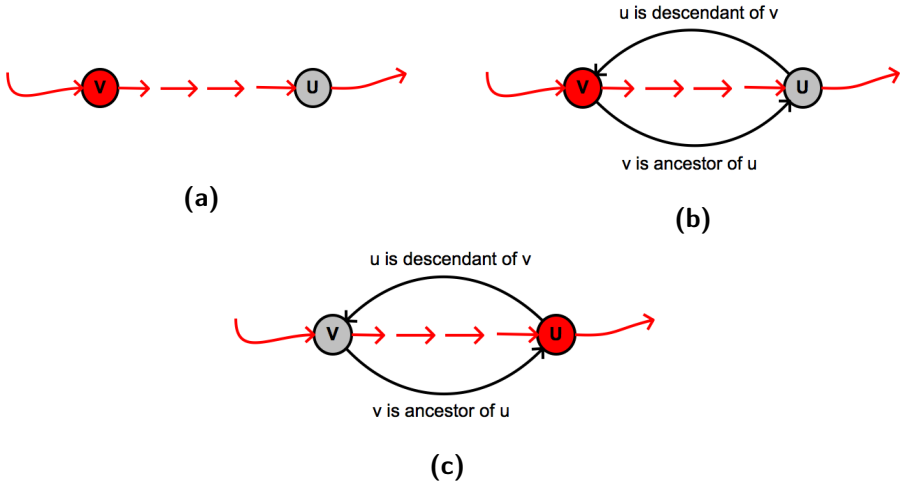


Figure 4.10: Example of how the descendants and ancestors are handled, described in the text.

we are exploring node v, and one of its neighbour, u, is already inserted in the tree. Moreover, both v and u have the same level. The possible outcomes are two:

1. u has been already explored before v: this means that v is a descendant of u and nothing should be done
2. u has been visited, but not yet explored: this means that u is, up to that moment, neither a relative nor a descendant of v

Figure 4.10 shows what must happen. The first node to be explored is v (Ariadne's tree, in red, imposes an ordering during the exploration), and one of its adjacent node, u, is already in the AD-tree (Figure 4.10a). Since u does not belong to the descendants and ancestors of v, it means that v precedes u in the exploration and v should be appointed as ancestor of u and, vice versa, u has to be included into the descendants of v (Figure 4.10b). When reaching node u during the exploration, it finds v as a candidate descendant, but since u is already connected to v in the descendants graph, nothing is done (Figure 4.10c).

These considerations highlight a peculiar property of the AD-tree, not disclosed in the previous subsection. Each node of the tree is self conscious of its location inside the tree itself. For a normal tree, a node only knows that it has been explored after its parent, grandparent, and all the nodes that can be found in the path connecting it to the root. Let's say that nodes belonging to different branches are cousins; then, the vertices of the AD-tree know their exploration order also with respect to their cousins.

Consider Figures 4.9 and 4.9c, representing a graph and its corresponding tree. Node 3 knows that it has been explored after nodes 1 and 0, but nothing

more: what about nodes 2 or 9, that are at the same level? Both Ariadne's tree and the descendants graph improve the connectivity scope of the node. From the former it knows exactly the nodes that are explored before and after it while from the latter it acknowledges the presence of nodes belonging to other branches.

Looking at Figure 4.9b, node 3 realizes that it is explored after 2 and before 6, but also after 7 and before 4, considering a range of one connected node. The expressed concept is extremely easy but very important: in a standard tree a node is equipped only with a vertical knowledge (parent, grandparent, children, grandchildren), but in the AD-tree the same node knows all the positions of the elements of the tree itself, exploiting the two enhancing data structures.

Lastly, the complexity of the construction algorithm should be considered. All the operations in the if and else branches in the main loop (lines 10 and 18 of Algorithm 6) are done in constant time, and it is clear that for each pass, the complexity is dependant on the degree of the node considered. Since exactly $|V|$ iterations are done, because we move along Ariadne's tree that includes all the nodes in the graph, the complexity is derived in the same way as breadth first and is

$$O(|V| + |E|)$$

4.3.2 *Construction example*

Consider the graph in Figure 4.11, and let node 0 be the root of the AD-tree. The root has three adjacent nodes, that are inserted in the tree as its children (we adopt a lexicographic order of insertion) and linked with Ariadne's tree, represented as a series of red arrows (Figure 4.12b).

The next node considered is 1, successor of node 0 in Ariadne's tree. The free adjacent nodes of 1 are nodes 2 and 3 (node 1 is already in the tree, and is its parent), that are consequently inserted as its children (Figure 4.12c). Node 5 follows, visiting nodes 6 and 9 (Figure 4.12d). Node 7 is the first explored node that has adjacent nodes already inserted: 3 and 6. Since their level is greater then the level of node 7, they are considered its descendants (blue arrows), and 7 is the ancestor of both of them (Figure 4.12e).

The steps repeat in the same way for the following nodes. Node 2 has one free child, 4, and one descendant, 9, that is at the same level (Figure 4.12f). The only child becomes the new tail of Ariadne's tree and the next node is considered. 3 has no free children, but has an ancestor, already determined when exploring node 7, and a descendant, node 4 (Figure 4.12g). Follows node 6, that has one free child, 8, and an ancestor, 7 (Figure 4.12h).

Node 9 is in the same situation as node 3: no free children, one ancestor and one descendant, 8 (Figure 4.12i). Node 4 is the last but one node to be

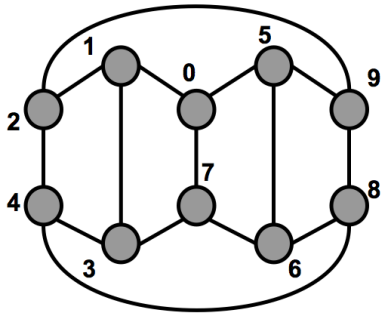


Figure 4.11: Graph considered for the AD-tree construction example.

explored, once again having all of its adjacent nodes already inserted into the tree, either as an ancestor, node 3, or as a descendant, node 8 (Figure 4.12i). Lastly, node 8 is reached and the exploration ends, since the node is also the last element of Ariadne's tree, meaning that all the nodes in the graph have been inserted in the AD-tree.

4.3.3 *Exploration variants*

The presented algorithm explores nodes in the classic breadth-first fashion. Since a lexicographic ordering is imposed, the exploration can be visualised as top-down, from left to right. Other three variations are now described, each with pros and cons.

4.3.3.1 *RtL AD-tree*

The Right to Left AD-tree is built as the normal AD-tree, but with inverse lexicographic order (for this reason, the latter is also called Left to Right AD-tree). While it may seems equivalent to the standard AD-tree, the generated structure may differ heavily due to the topology of the associated graph. Considering the graph used in the example before (Figure 4.11), the corresponding RtL AD-tree is represented in Figure 4.13

The RtL AD-tree is computed easily and as the standard structure it suffers of two problems: first its node distribution is uneven among the branches; second, the minimum and maximum distances from the root to the leaves can differ greatly.

4.3.3.2 *Alternated AD-tree*

Instead of picking a fixed lexicographic order, the alternated AD-tree inverts it at each level. For example if, as in Figure 4.14, level 1 has a left to right order, level 2 will have a right to left order, and so on. This version of the AD-tree is the most difficult to implement, because the corresponding Ariadne's

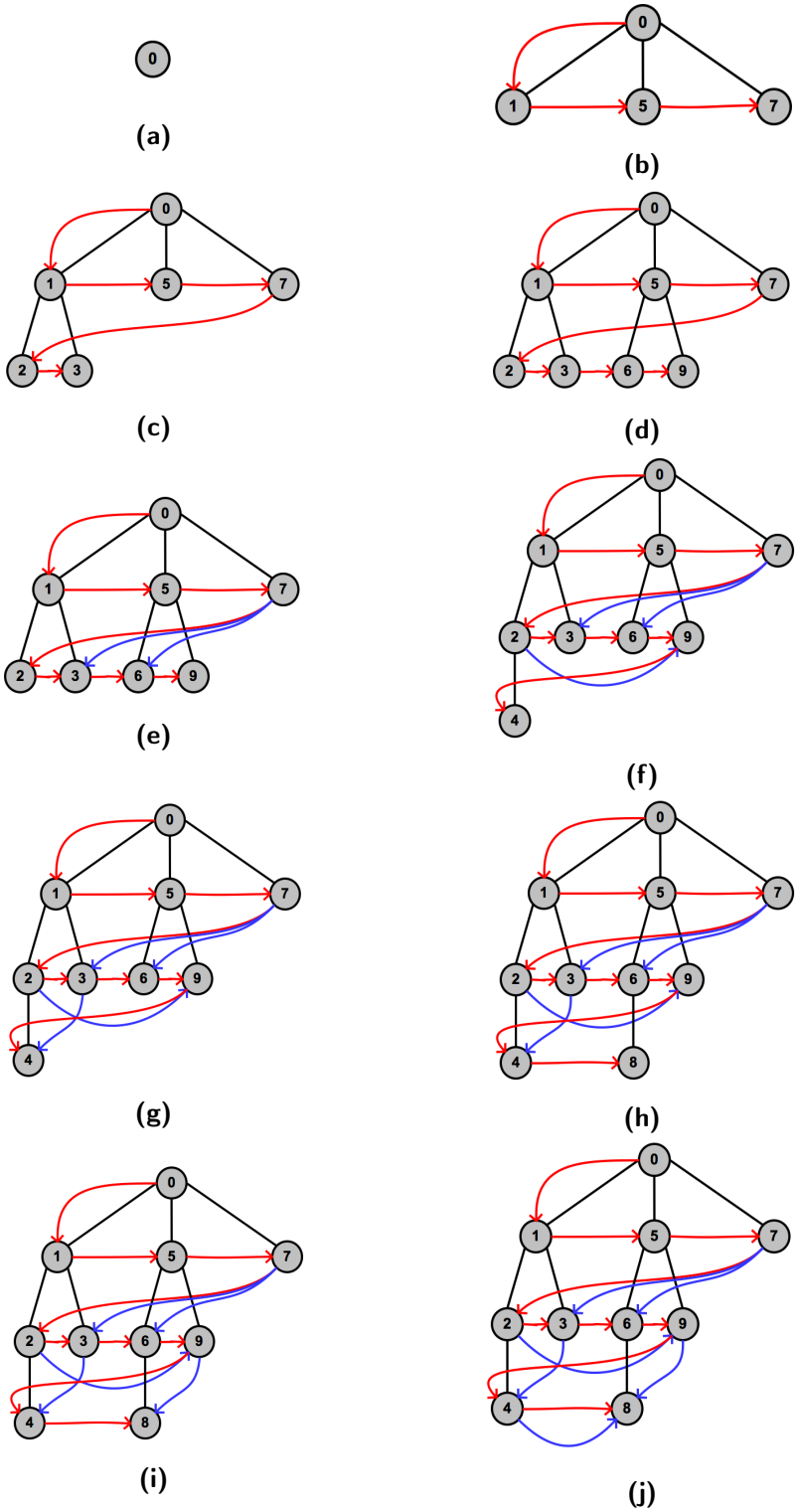


Figure 4.12: Construction of the AD-tree for graph showed in Figure 4.11.

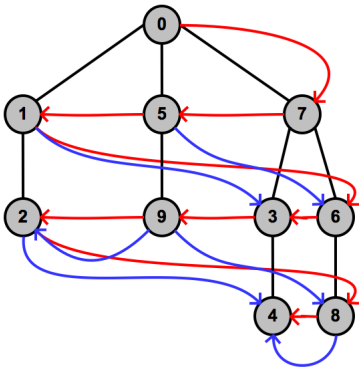


Figure 4.13: RtL AD-tree of the graph showed in Figure 4.11.

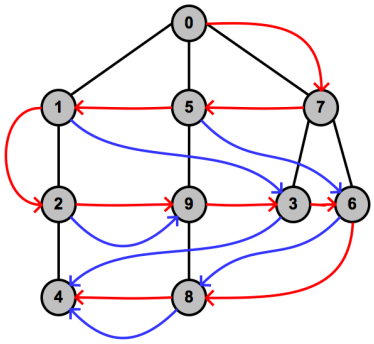


Figure 4.14: Alternated AD-tree of the graph showed in Figure 4.11.

tree cannot be used directly when exploring nodes. The adopted solution resembles the way DNA is replicated on the lagging strand: at each level of the AD-tree, nodes are linked through pieces of Ariadne's tree, that are later combined to form the whole structure.

For example, in Figure 4.14 mentioned before, we start with a inverted order, hence the first part of Ariadne's tree consists of nodes {0,7,5,1}, in this order. 7 is the first node to be expanded, with children 3 and 6. They are connected to form the Ariadne's tree fragment of the second level. Then 5 visits 9, and 1 visits 2, leading to the fragment {6,3,9,2} (moreover, they have descendants already in the tree, respectively nodes 6 and 3). After node 1, no node is available to be explored. This starts the merging phase, that consists in inverting the sequence of edges of the found fragment ({2,9,3,6}), and chaining the last explored node and the first of the inverted fragment. Now, the partial Ariadne's tree is formed by the series of nodes {0,7,5,1,2,9,3,6}. The process repeats in the same way: 2 visits 4, becoming ancestor of 9, and 3 becomes parent of node 8 (having 4 as a descendant), creating fragment {4,8}. Nodes 3 and 6 have to children but only descendants. The fragment

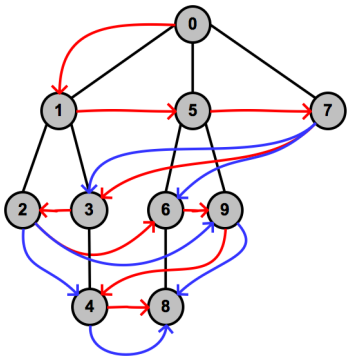


Figure 4.15: Flippant AD-tree of graph showed in Figure 4.11.

is then inverted and attached to the remaining Ariadne's tree, obtaining the desired snake-like structure.

Although the alternated AD-tree is harder to build, it provides a structure with more evenly distributed nodes and shorter paths from the root to the leaves.

4.3.3.3 *Flippant AD-tree*

The RtL AD-tree inverts the visit/expansion ordering once for all the structure; the alternated AD-tree does the same thing but once at each level of the tree. The flippant AD-tree changes the lexicographic ordering of nodes whenever a node is explored, meaning that it changes $|V| - 1$ times.

Considering the graph in Figure 4.11, the expansion and visiting follow the usual rules. Node 0 has children 1, 5 and 7, connected together and expanded in this order. The first node expanded is 1, with heirs 2 and 3. Since the first expansion (of node 0) had ascending order, now we link nodes 2 and 3 in descending order to the rest of Ariadne's tree. The next node to be explored is 5, with children 6 and 8. The order returns ascending, so the nodes are inserted as presented in the last sentence, and so on (Figure 4.15).

The flippant AD-tree is a compromise between the RtL/LtR tree and the alternated tree. It is slightly harder to implement than the first, but provides a good distribution of nodes, similarly to the latter.

4.3.4 *Parallel construction algorithm*

The construction of an AD-tree can be parallelised in a clever way: instead of applying the procedure described in Algorithm 6 starting from the root of the tree, children nodes can be used as temporary roots to be explore in parallel. This breaks the algorithm into two phases. First, it starts exploring nodes from the root in serial, as described before. Then, instead of exploring

Algorithm 7 AD-tree linking

```
1:  procedure ADTreeLinking(root)
2:      initialise Ariadne's tree cursor atc ← root
3:      initialise Ariadne's tree tail cursor attc ← root
4:      while atc points to an existing node do
5:          current node v ← atc
6:          for all v_c ∈ v.children do
7:              v_c.prev ← attc
8:              attc.next ← v_c
9:              attc ← v_c
10:         end for
11:         atc ← atc.next
12:     end while
13: end procedure
```

the whole set of nodes in the graph, it stops at a certain level of the tree. All the nodes belonging to that level will be considered temporary roots, and the tree creation will proceed separately on each of them.

For example, in Figure 4.16, the standard LtR AD-tree construction algorithm is adopted, beginning with node 0. After having filled the first level, the serial phase is interrupted and each of the elements in the level are now considered separate roots. The building procedure is then used separately on nodes 1, 5 and 7, having attention to not insert nodes already present into another branch of the main tree (meaning that some degree of communication between processors is needed).

Parallelisation makes the construction faster, but requires two additional steps. First, as it can seen in the example of Figure 4.16, Ariadne's tree is built incorrectly, letting them have more than one children, while the correct tree consists in nodes with only one child and one parent, except the root and the last node. For this reason the whole tree has to be revisited, from root to leaves, and connect the nodes as if they were built in the serial way. This procedure is very fast because it only needs to explore nodes, that are already existent, in the tree.

What presented in Algorithm 7 is the left to right approach (following lexicographic order) to reconstruct the Ariadne's tree, but the same variants described previously for building the AD-tree can also be repeated here, obtaining multiple combinations. For example, the AD-tree can be constructed left to right, but Ariadne's tree is recreated right to left on the already built AD-tree. Moreover, the algorithm allows to change the Ariadne's tree at any time, even after the AD-tree is complete.

The other problem is related to descendants and ancestors, that cannot be inserted while the construction algorithm is ongoing, due to concurrency

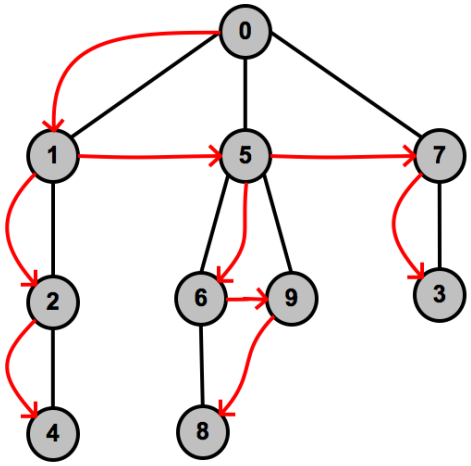


Figure 4.16: One of the possible parallel AD-trees of graph showed in Figure 4.11.

reasons. The solution is straightforward: mark nodes that should be part of a descent/ancestry relationship and, after the tree is built, connect them.

Overall, the parallel construction decreases the computational time required to form an AD-tree and also allows an even distribution of nodes among the branches. However, it requires further steps, differently from the serial algorithm where all the components of the AD-tree are built simultaneously.

4.4 AD-TREE PARTITIONING

In this section, the partitioning algorithm of the AD-tree will be described, along with an example on a small graph.

4.4.1 *Partitioning algorithm*

Starting from the last node of Ariadne's tree, that is the node with no next element, the tree is backtracked and each node propagates the size of rooted subtree in it to its parent, until cutting conditions are met. Precisely, having a tree of $|V|$ nodes that should be partitioned into k sets, the tree is cut when a node is the root of a subtree having about $|V|/k$ elements. The rooted subtrees are not to be intended as composed only by children and grandchildren, but augmented also with descendants. Propagations and cuts continue until the $(k-1)^{th}$ partitioning is done: the last node to cut will surely be the root of the tree.

The main body of the partitioning algorithm, showed in Algorithm (8, consists in a series of conditions that are checked on the tree nodes.

Algorithm 8 AD-tree partitioning main body

```
1: procedure ADTreePartitioning(root,k)
2:     initialise current node pointer cnp with the last element of Ariadne's tree, computed starting from root
3:     initialise cut threshold cutThresh ← |V|/k
4:     current partition cp ← 0
5:     while cnp points to a node do
6:         cnp points to current node cn
7:         if cp = k − 1 then
8:             CutSubtree(root,cp)
9:             end the algorithm
10:        end if
11:        if cn.isValid = false then
12:            cnp ← cnp.prev
13:            jump to next iteration
14:        end if
15:        if cn.value ⩾ λ ∗ cutThresh then
16:            SearchDescendants(cn,cp,cutThresh)
17:            if the previous call ended with a cut then
18:                cp ← cp + 1
19:                cnp ← cnp.prev
20:                jump to next iteration
21:            end if
22:        end if
23:        if ∑_{i=0}^{|cn.parent.children|} (cn.parent.children[i].value) ⩾ ϵ ∗ cutThresh, with cn.parent.children[i] preceding cn in Ariadne's tree then
24:            CutSubtree(max between these cn.parent.children,cp)
25:            cp ← cp + 1
26:            if cn is the cut node then
27:                cnp ← cnp.prev
28:                jump to next iteration
29:            end if
30:        end if
31:        if cn.value ⩾ α ∗ cutThresh then
32:            CutSubtree(cn,cp)
33:            cp ← cp + 1
34:        else
35:            Propagate(cn)
36:        end if
37:        cnp ← cnp.prev
38:    end while
39: end procedure
```

- If $k-1$ partitions have already been created, it means that the remaining nodes form the last partition. Consequently there is no need to continue backtracking along Ariadne's tree and the root of the tree can be immediately cut, ending the algorithm (line 7 of Algorithm 8). If less than $k-1$ partitions are identified, the algorithm continues.

- If the current node is invalid (line 11 of Algorithm 8), meaning that is already assigned to a partition, all the other checks can be bypassed ignoring the node and moving to the next one (that is the previous in Ariadne's tree, current node prev). If the current node is valid, the algorithm continues.

- If a node value, meaning the number of nodes in the rooted subgraph in it, is greater than a certain number $\lambda * \mathtt{cutThresh}$, with $0 < \mathtt{lambda} < 1$, the descendants of that node are searched. This (from line 15 of Algorithm 8) has the goal to find a series of consecutive descendants (a has b as descendant, that has c as descendant, that has d as descendant, and so on) such that the sum of their values and the node value is greater than the cut threshold, identifying a new partition. The search is done with a variant of depth first, because the algorithm searches for paths (beginning in the current node and ending in a descendant or descendant of descendant, and so on). If the search ends successfully and a cut is performed, the algorithm moves to the next node, otherwise it continues.

- If the sum of the current node value and the values of its valid siblings (nodes with the same parent) that have yet to be traversed by the algorithm exceeds the cut threshold by a factor $\epsilon > 1$, the node having maximum value is cut, upper bounding the dimension of the created partitions (step at line 23 of Algorithm 8). In particular if the cut node is the current one, the algorithm jumps to the successive iteration, considering the next node to be traversed. Otherwise, it continues.

- Lastly, if the current node itself has value slightly greater than the cut threshold by a factor α (at line 31 of Algorithm 8), it is cut to form a new partition. If a cut cannot yet be performed, the node's value is propagated to the parent, and the next node is considered, ready to begin a new iteration of the algorithm's main cycle.

Each subroutine called by the body of the procedure is now be described. The CutSubtree subroutine (Algorithm 9) is straightforward: in a depth-first fashion, it visits all the nodes of a subtree and assigns them to a partition. It should be pointed out that although the procedure is called CutSubtree, there is no real cut in the AD-tree data structure, but simply a colouring

Algorithm 9 CutSubtree routine

```
1: procedure CutSubtree(root,currPartition)
2:     initialise stack of nodes NS
3:     insert root into NS
4:     while NS is not empty do
5:         current node cn ← NS.top
6:         remove NS.top
7:         cn.group ← currPartition
8:         mark cn as invalid
9:         for all child ∈ cn.children do
10:            if child is valid then
11:                insert child into NS
12:            end if
13:        end for
14:    end while
15: end procedure
```

Algorithm 10 Propagate routine

```
1: procedure Propagate(v)
2:     mark v as propagated
3:     v.parent.value ← v.parent.value + v.value
4: end procedure
```

of the nodes, so that no connection is altered. During the main algorithm execution each node goes through this procedure exactly once.

The second routine is Propagate (Algorithm 10), consisting only of two instructions: mark the node as propagated and add its value to its parent's value.

The third and last routine is the most complex. Given a node, if descendants are present, they define a tree. Consider the example used to show how the construction algorithm works and focus on the complete AD-tree, in Figure 4.12j. Node 7 has associated a small tree of descendants, consisting of nodes 3 and 6 for the first level, node 4 for the second and node 8 for the last level.

The SearchDescendants procedure (Algorithm 11) searches for a path, starting from the root of the descendants graph of a node, that actually is the node itself, such that the sum of the values of all the nodes in the path is greater than the cut threshold, meaning that they can form a new partition.

The procedure uses two stacks: one contains the candidate roots to be cut, while the other contains stacks of explored descendants, that is the main difference from classic depth first. The stack of stacks is needed because during the procedure there is the need to know which descendant is currently being

Algorithm 11 SearchDescendants routine

```
1: procedure SearchDescentands(v,currPartition,cutThresh)
2:     initialise container DS (descendants stack of stacks)
3:     initialise container CS (stack of nodes to cut)
4:     put v into DS as a single element stack
5:     put v into CS
6:     for all des ∈ v.descendants do
7:         if des is valid then
8:             put des in a temporary stack TS
9:         end if
10:    end for
11:    put TS into DS

12:    while DS is not empty do
13:        if DS.top is empty then
14:            remove DS.top
15:        end if

16:        if DS.top.top = CS.top then
17:            remove DS.top
18:            remove CS.top
19:        end if

20:        current descendant cDesc ← DS.top.top
21:        put cDesc into CS
22:        if ∑_i CS(i) > α * cutThresh then
23:            for all node ∈ CS do
24:                CutSubtree(node,currPartition)
25:                DepleteSubRoot(node,v.level)
26:                stop the procedure
27:            end for
28:        end if

29:        for all des ∈ cDesc.descendants do
30:            if des is valid then
31:                put des in a temporary stack TS
32:            end if
33:        end for
34:        put TS into DS
35:    end while
36: end procedure
```

Algorithm 12 DepleteSubRoot routine

```
1:  procedure DepleteSubRoot(v,stopLvl)
2:      while v.level <= stopLvl do
3:          if v is propagated then
4:              v.parent.value ← v.parent.value − v.value
5:              v ← v.parent
6:          else
7:              stop the procedure
8:          end if
9:      end while
10: end procedure
```

explored at each level. In fact, considering the recurrent case of failed search (a path root-leaf is explored but the sum of its nodes values is not enough to start the cutting phase), the procedure needs to remove some of the candidate nodes. This implementation allows a connection between the two data structures: the single stack of candidates contains the tops of each stack in the other structure. When a search fails, there is a correspondence between the top elements, so that the node candidates can be easily handled.

A small example of how the procedure works is represented in Figure 4.17, referring to the small descendants tree rooted in node 7, in Figure 4.12j. Moreover, in the example the search is supposed to fail, leading to no cut. The two main structures, DS (descendants stack of stacks) and CS (stack of nodes to cut) are filled with the tree root, 7 (Figure 4.17a). Then, the descendants of 7 are inserted in DS, but only its top node candidates for a cut. So, node 3 is put in CS and will be the next node to be explored (Figure 4.17b). Node 4 is its only descendant, inserted both in DS and CS (Figure 4.17c), and the same goes for 8 (Figure 4.17d). Since it has been assumed that all the searches fail, added to the fact that 8 has no descendant, the algorithm begins to pop nodes from the two main structures. 8 is the first (Figure 4.17e), followed by 4, both in DS and CS (Figure 4.17f). Node 3 is popped and substituted by node 6 in CS. (Figure 4.17g). 6 has no descendants and, after having emptied DS and CS two more times, the procedure ends (Figure 4.17h).

When a search is successful, after the cut, it is also necessary to remove the values of already propagated nodes, because they no longer contribute to the rest of the AD-tree. This is done in the DepleteSubRoot procedure (Algorithm 12).

Before making an example of how the AD-tree partitioning algorithm works, few words must be spent about the meaning and assignments of the tree parameters in it.

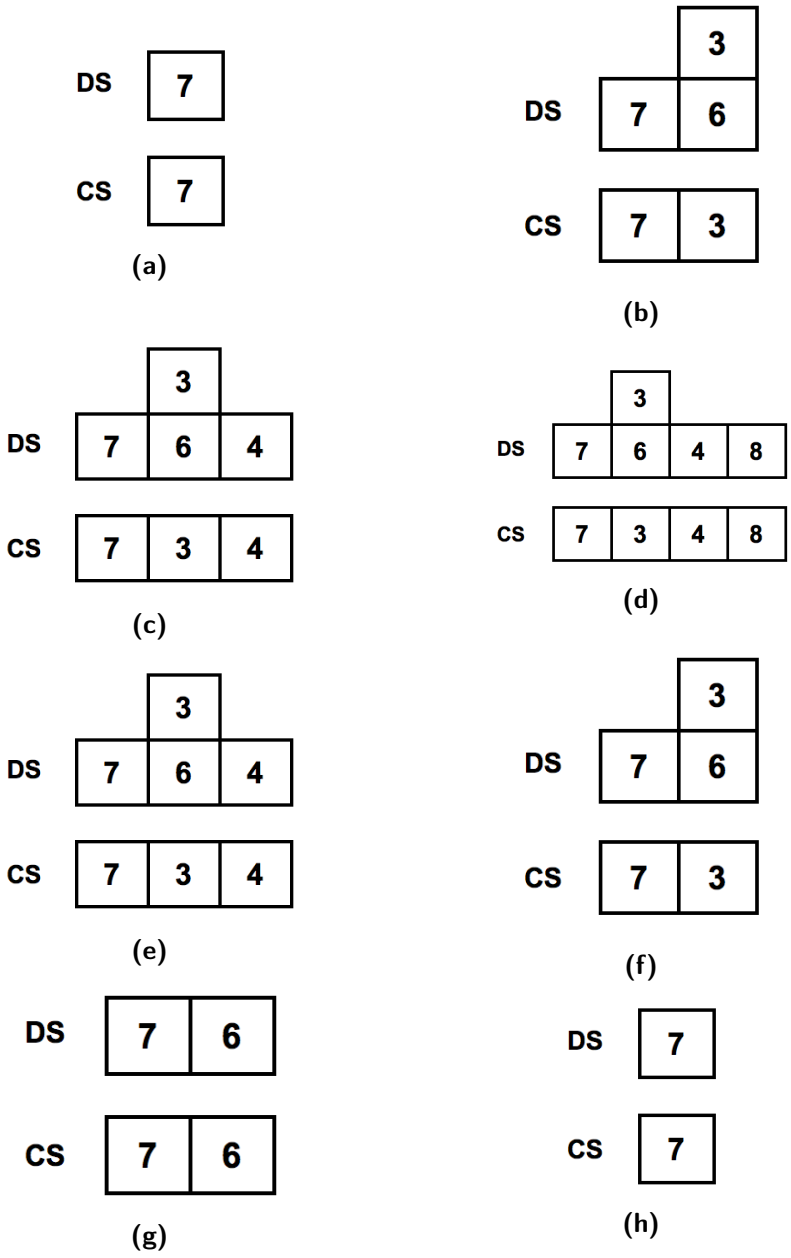


Figure 4.17: Example of how the SearchDescendants procedure works, described in the text.

- λ, usually fixed to 0.33, is a dampening factor of the cut threshold, marking a node available for the SearchDescendants procedure (Algorithm 11). If a node value is greater than the quantity $\lambda \cdot \mathtt{cutThresh}$, the procedure is called. The purpose of this parameter is to avoid calling the routine unnecessarily, in the early iterations of the AD-tree parti-

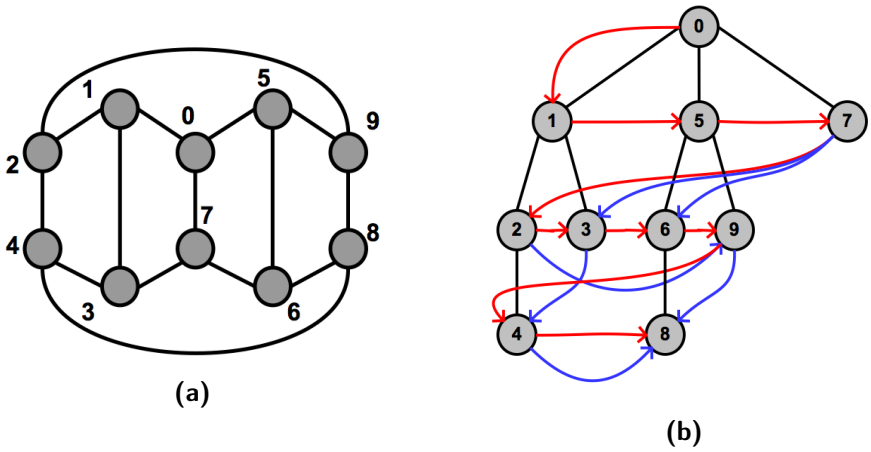


Figure 4.18: Graph and corresponding AD-tree used in the algorithm demonstration.

tioning algorithm (Algorithm 8), when the node values are still to low to perform a cut.

- ϵ determines the upper bound of the partitions, meaning that no group (except the last one, that is formed by the remaining nodes of the tree) should exceed in size the quantity $\epsilon \cdot \mathtt{cutThresh}$; it is usually set to 1.3 or 1.4.
- α grants some unbalancing in the partitions, allowing a cut with size different from the cut threshold; it is greater than 1 and lesser than ϵ

4.4.2 *Partitioning example*

To see in practice the partitioning algorithm, the graph represented in Figure 4.18a is considered, along with the corresponding LtR AD-tree, of Figure 4.18b. Suppose that the tree should be partitioned into 5 sets, meaning that each should have 2 nodes in it. At the beginning of the algorithm, all the nodes have value equal to 1, since no traversing has been done. The parameters are fixed in the following way: $\lambda = 0.4$, $\epsilon = 1.4$ and $\alpha = 1$.

Node 8 is the last of Ariadne's tree and the first to be considered in the partitioning algorithm (Figure 4.19a). It is valid, it does not exceed the cut threshold and is an only child: it can only propagate to the parent, increasing its value (Figure 4.19b). The next node to be considered is 4 (Figure 4.19c). The value of this node enables the SearchDescendants procedure (being $1 > \lambda \cdot \mathtt{cutThresh} = 0.4 \cdot 2 = 0.8$). The only descendant is 8, and together they reach the cut value (being $2 \geqslant \alpha \cdot \mathtt{cutThresh} = 1 \cdot 2 = 2$). The two nodes are cut and the value of node 8 is taken away from its parent (Figure 4.19d).

The backtracking continues with node 9, the first node with a sibling (Figure 4.19e). Together they do not surpass the max size bound $\epsilon * \texttt{cutThresh} = 1.4 \cdot 2 = 2.8$, hence 9 is just propagated to the parent. The same is done for node 6, propagated to the same parent (Figure 4.19f) and node 3, whose value is added to the value of node 1 (Figure 4.19g). However, node 2 (Figure 4.19h) is eligible for the SearchDescendants procedure, that pairs it with node 9, forming a new group (Figure 4.20a).

The value of node 9 is subtracted to its parent value and the new pass of the algorithm begins with node 7 (Figure 4.20b). It has two descendants, nodes 3 and 6, but already considering the first, the cut value is reached, hence 7 and 3 form a new partition (Figure 4.20c). The first is not propagated to the parent, while the value of the latter is taken away from node 1.

The new iteration sees 5 as its current node (Figure 4.20d). The node is valid and has no descendants, but has node 1 as a sibling. Together their values exceed the max size acceptable for a partition ($1 + 2 > 2.8$) and the node with max value is cut: 5. Since node 5 has only one free child, they form the fourth partition (Figure 4.20e).

The last but one node to be considered is now 1 (Figure 4.20f), but since the algorithm has already performed $k - 1$ cuts, there is no need to do any check: the root, node 0, is directly cut and assigned to the last partition, containing also node 1 (Figure 4.20g).

The partitions obtained in this example are perfectly balanced, but this is due to the small dimension of the original graph. In real situations, considering structures containing from ten thousands to millions of nodes, the proposed algorithm achieves reasonable, and not perfect, balance.

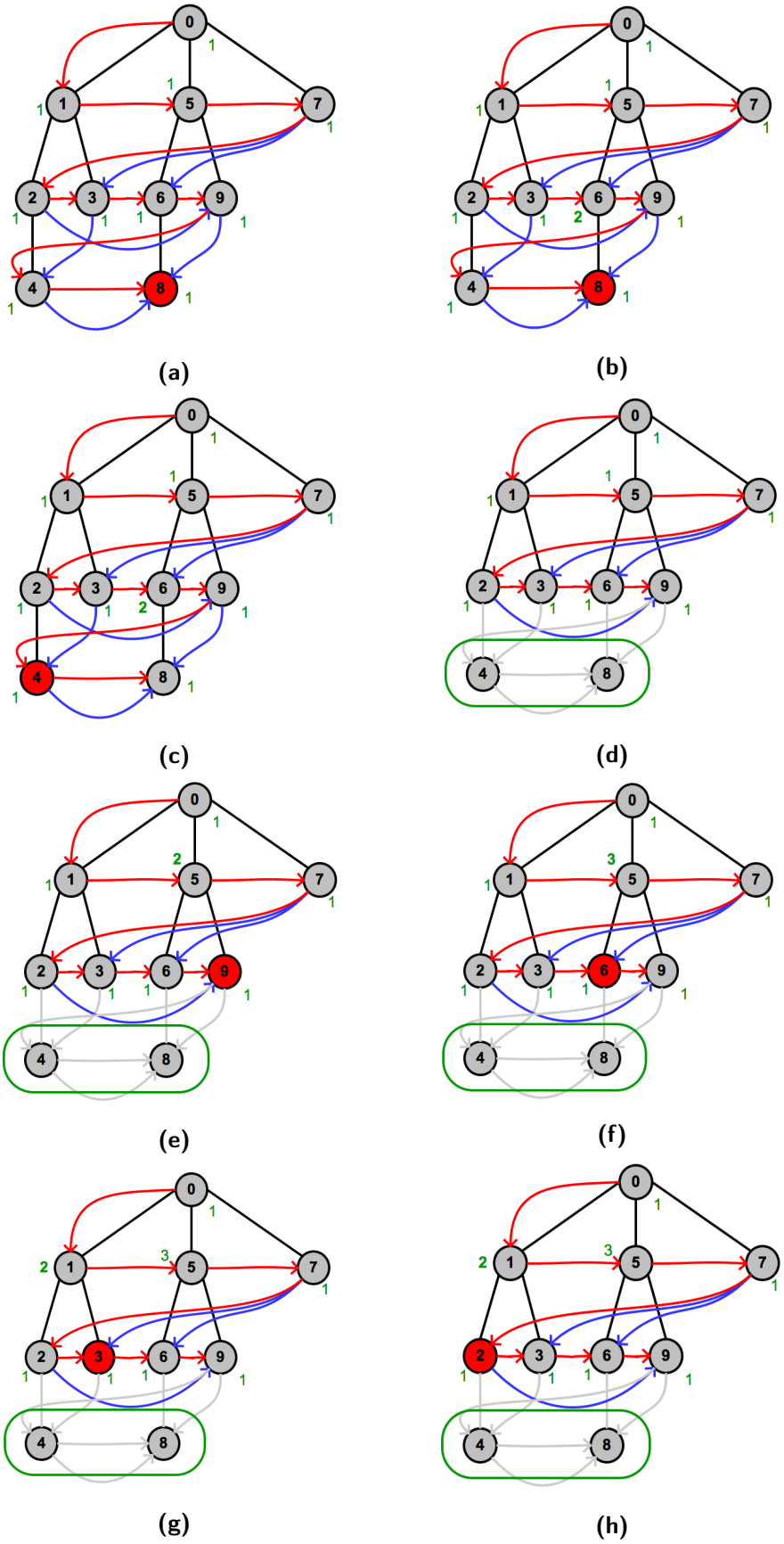


Figure 4.19: First part of the example describing how the partitioning algorithm works.

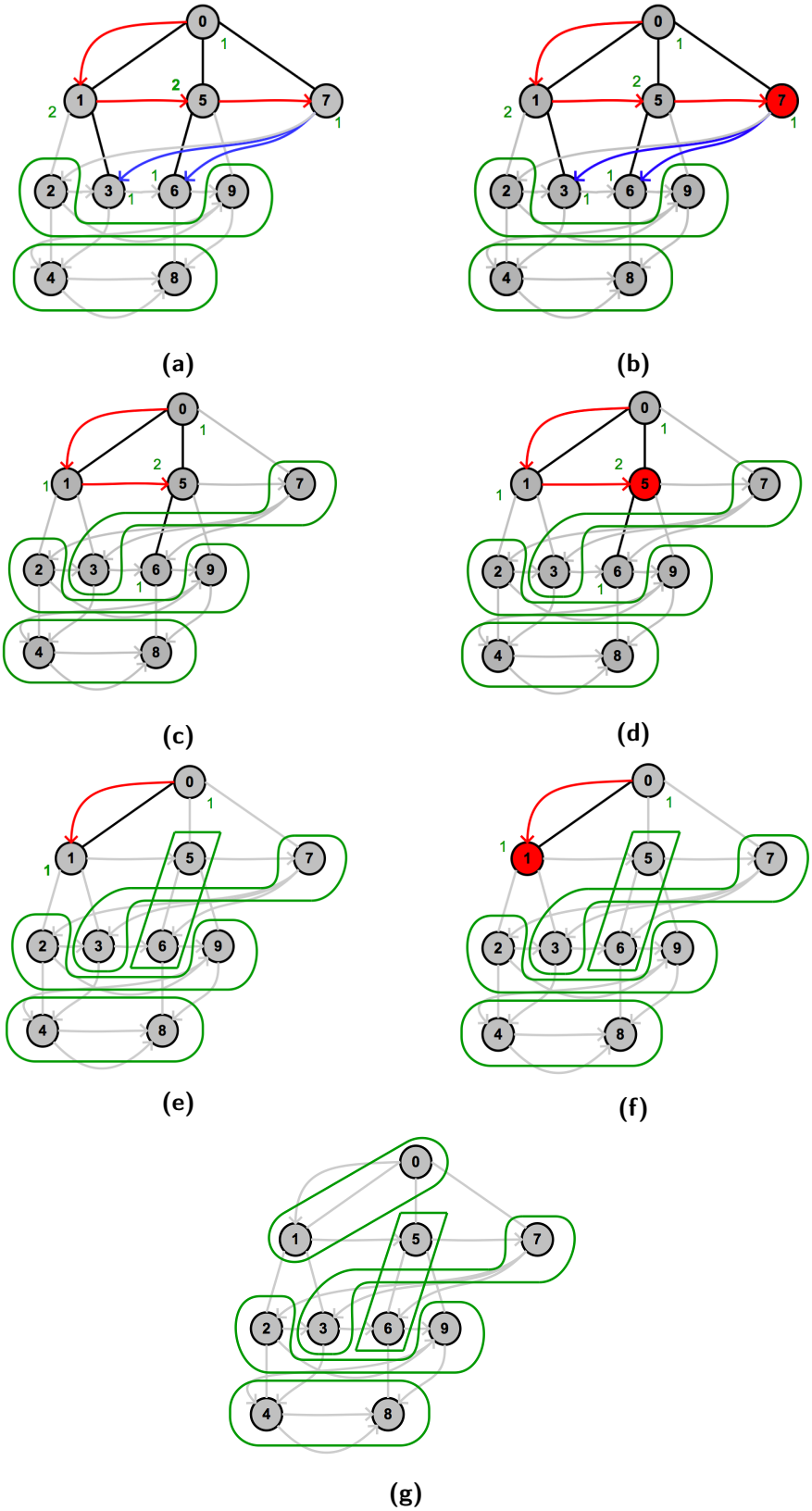


Figure 4.20: Second part of the example describing how the partitioning algorithm works.

4.5 AD-TREE PARTITIONING RESULTS

To demonstrate the results of the AD-tree partitioning algorithm, we use the graphs derived from the mesh collection provided with the benchmark described in [4] (considering their faces as nodes), comparing them with the partitioning obtained from Zoltan RCB [75], Zoltan RIB [75], Zoltan HSFC [75], Zoltan PHG [74], KaFFPa [39], METIS [38, 76] and ParMETIS [64, 65, 63, 60], for $k = 8$.

The following configuration is used.

- Values are taken as average over different runs on a 64-bit laptop with Intel® Core(TM) i5-3337U CPU @ 1.80GHz x 4 processors, each with 3072Kb of cache size.
- All the algorithms from the Zoltan toolkit are set to work in parallel using as many threads as k.
- ParMETIS uses the *ParMETIS_V3_PartMeshKway* routine to compute the partitions directly on the surface mesh. Default parameter values are used, as suggested in the toolkit manual [60]. Only one thread is considered.
- METIS uses the *METIS_PartMeshDual* routine to partition a mesh into k parts based on a partitioning of its dual graph. Also here, default parameter values are used, following the toolkit manual indications [76].
- The AD-tree partitioning parameters are set as follows: $\lambda = 0.33$, $\epsilon \in \{10, 15, 20\}$, $\alpha = 15$. In figures, the best partitions are showed, while in the tables values are taken as average over the results corresponding to each ϵ.
- The AD-tree construction is done in serial and its time is computed as average over the tree possible implementations (LtR, RtL, Alternated and Flippant).

To evaluate the algorithms, three metrics are considered:

1. Execution time consisting in partitioning plus eventual data conversions (only for the proposed algorithm, that relies on the AD-Tree, showed in two lines: top represents tree conversion plus partitioning, bottom is their sum)
2. Maximum and minimum imbalances, useful to judge the ability of a procedure to obtain balanced partitions; they are respectively computed as

$$\frac{\max |\,\mathrm{partition}_i\,|}{|\,\text{perfectly balanced partition}\,|}$$

and

$$\frac{\min |\,partition_i\,|}{|\,\text{perfectly balanced partition}\,|}$$

3. The percentage of border nodes of a partition with respect to its size, defined as

$$\frac{\text{num border nodes of } partition_i}{|\,partition_i\,|}$$

Lower values of the three metrics correspond to high quality partitioning.

It should be noticed that the execution time presented includes data conversions only for the proposed algorithm. This does not mean that the other compared methods are good to use directly on the meshes. Zoltan, ParMETIS (with METIS) and KaFFPa requires each further step to convert the initial meshes to the format accepted by them, further increasing the execution time. We do not consider these times to strengthen the fact that our proposed algorithm, considering data conversion plus partitioning, is faster than their partitioning alone.

Figures 4.21, 4.22, 4.23, 4.24, 4.25, 4.26 and 4.27 represent the resulting partitions of the considered meshes, showing the outcome of each algorithm. As it can be seen by the figures, the geometric methods (RCB, RIB and HSFC), METIS and PHG generate disconnected partitions. KaFFPa, ParMETIS and our proposed method produces instead connected blocks. Also, to geometric methods correspond hard partitions, meaning that visually they are rectangular blocks (motivated by the fact that these methods partition the space and not the mesh itself). The other algorithms, including the one proposed, produce smooth and enveloping partitions.

Let's consider now the metrics associated to the meshes considered, described in Tables 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 and 4.7. AD-tree partitioning excels over all the other algorithms in terms of execution time (between round brackets), being about one third of the best performant method within them, ParMETIS. Moreover, if we consider only the partitioning time, we reach a result one order of magnitude cheaper than the state of the art. It is trivial to see that, for every algorithm, the time complexity is linear with respect to the number of nodes of the graph to partition.

In terms of unbalance, our algorithm classifies in the last but one place, right above METIS and below Zoltan PHG. It should be however noticed that the gap between AD-tree partitioning and METIS is quite large, making it acceptable compared to the rest of the algorithms (see Figures 4.28 and 4.29).

Lastly, consider Tables 4.8 and 4.9, representing the third metric, in percentage. With the exception of the mesh hand, whose AD-tree partitioning (Figure 4.25h) is quite peculiar, the ratio number of border elements and size of the considered partition is in the same range for all the algorithms.

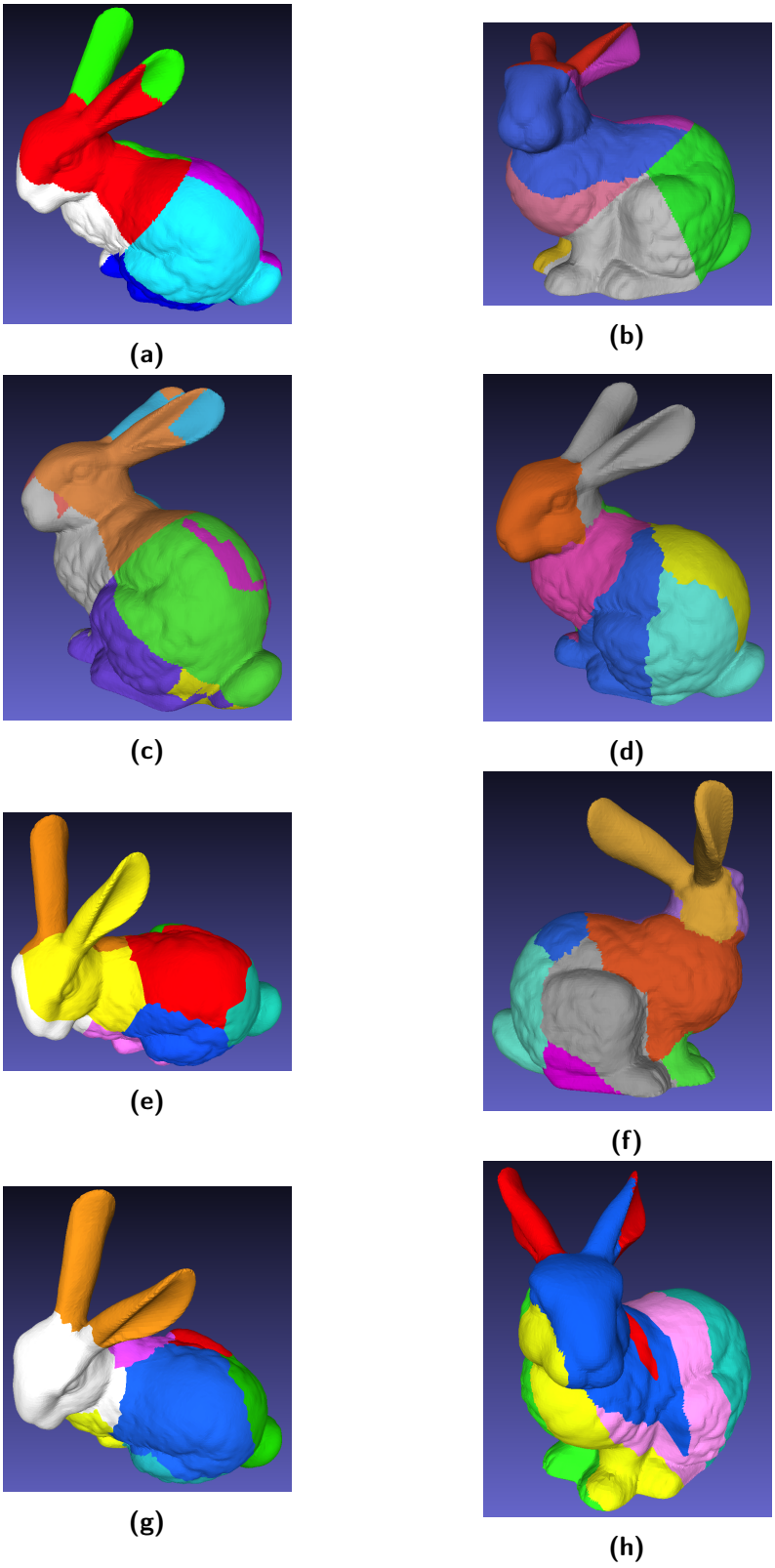


Figure 4.21: From top to bottom, left to right: Zoltan RCB, Zoltan RIB, Zoltan HSFC, Zoltan PHG, KaFFPa, METIS, ParMETIS and AD-tree partitioning on the mesh bunny.

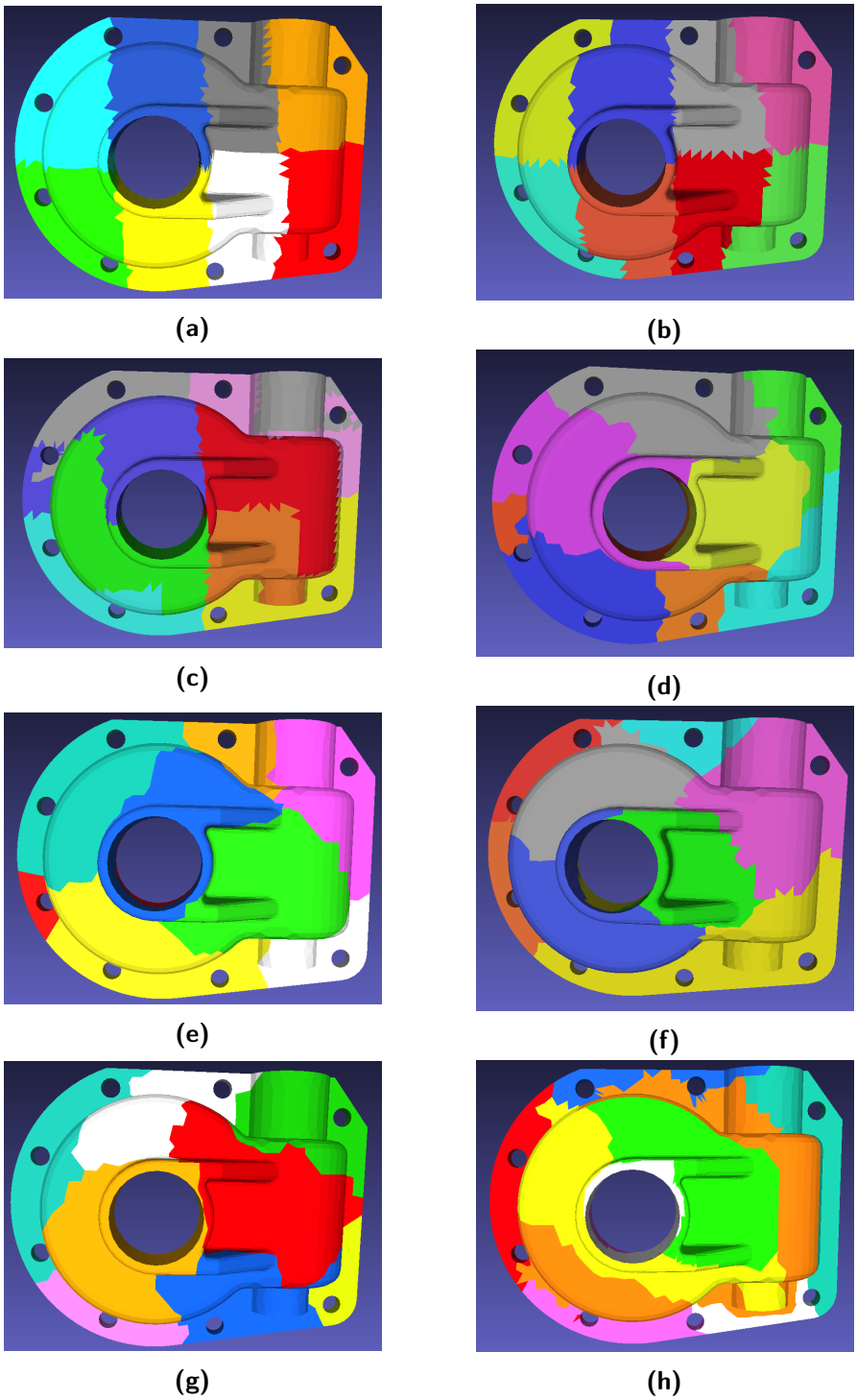


Figure 4.22: From top to bottom, left to right: Zoltan RCB, Zoltan RIB, Zoltan HSFC, Zoltan PHG, KaFFPa, METIS, ParMETIS and AD-tree partitioning on the mesh casting.

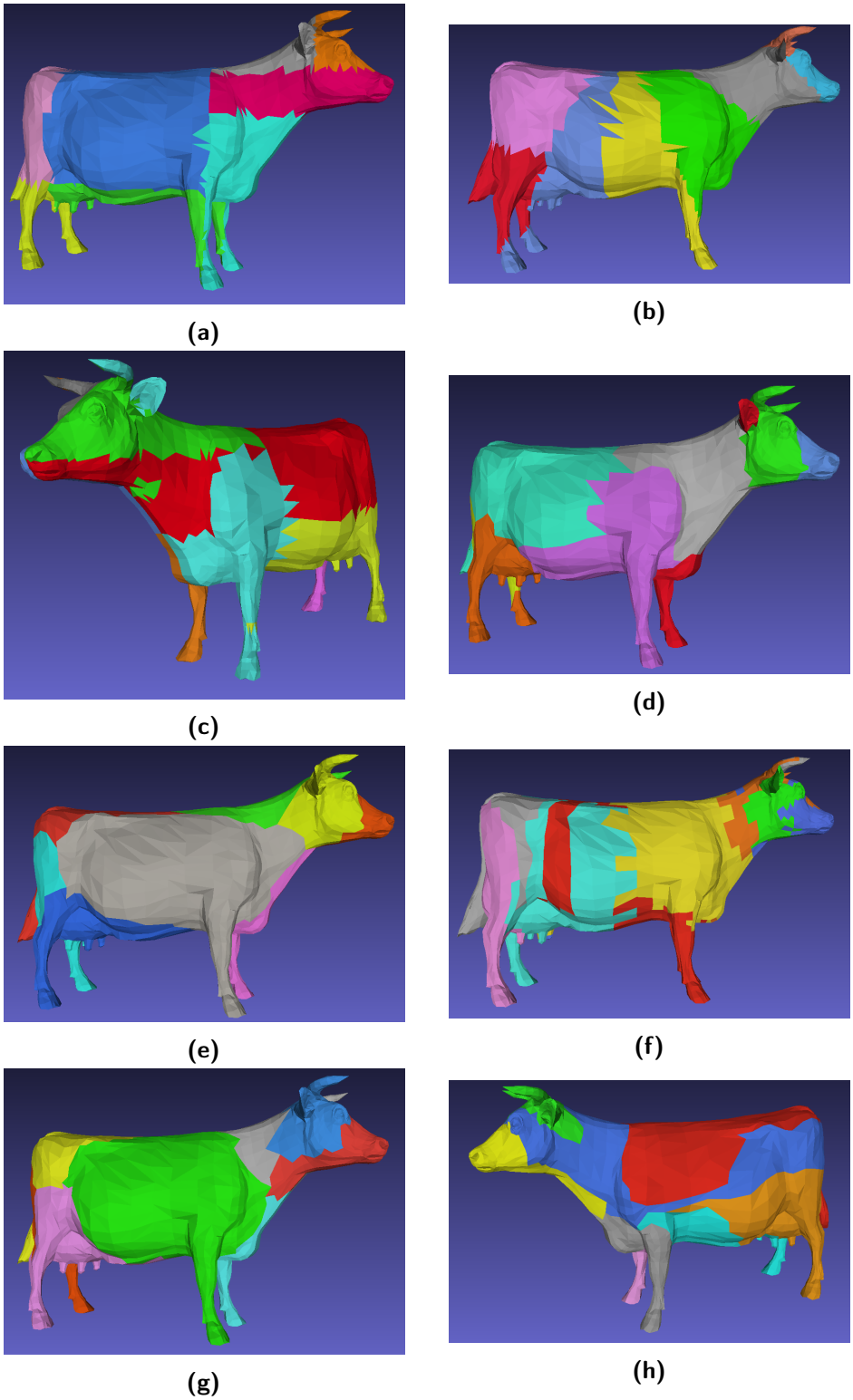


Figure 4.23: From top to bottom, left to right: Zoltan RCB, Zoltan RIB, Zoltan HSFC, Zoltan PHG, KaFFPa, METIS, ParMETIS and AD-tree partitioning on the mesh cow.

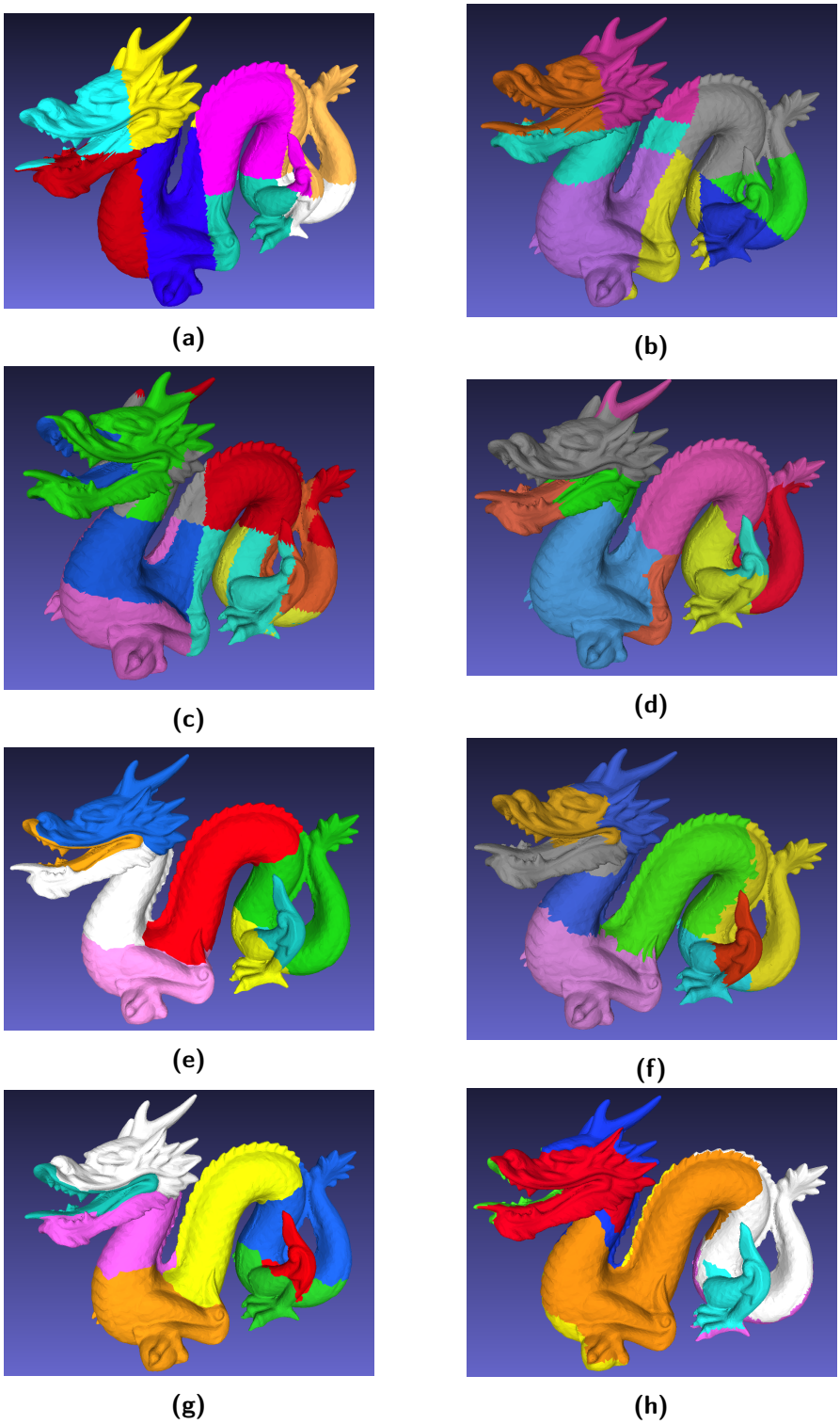


Figure 4.24: From top to bottom, left to right: Zoltan RCB, Zoltan RIB, Zoltan HSFC, Zoltan PHG, KaFFPa, METIS, ParMETIS and AD-tree partitioning on the mesh dragon.

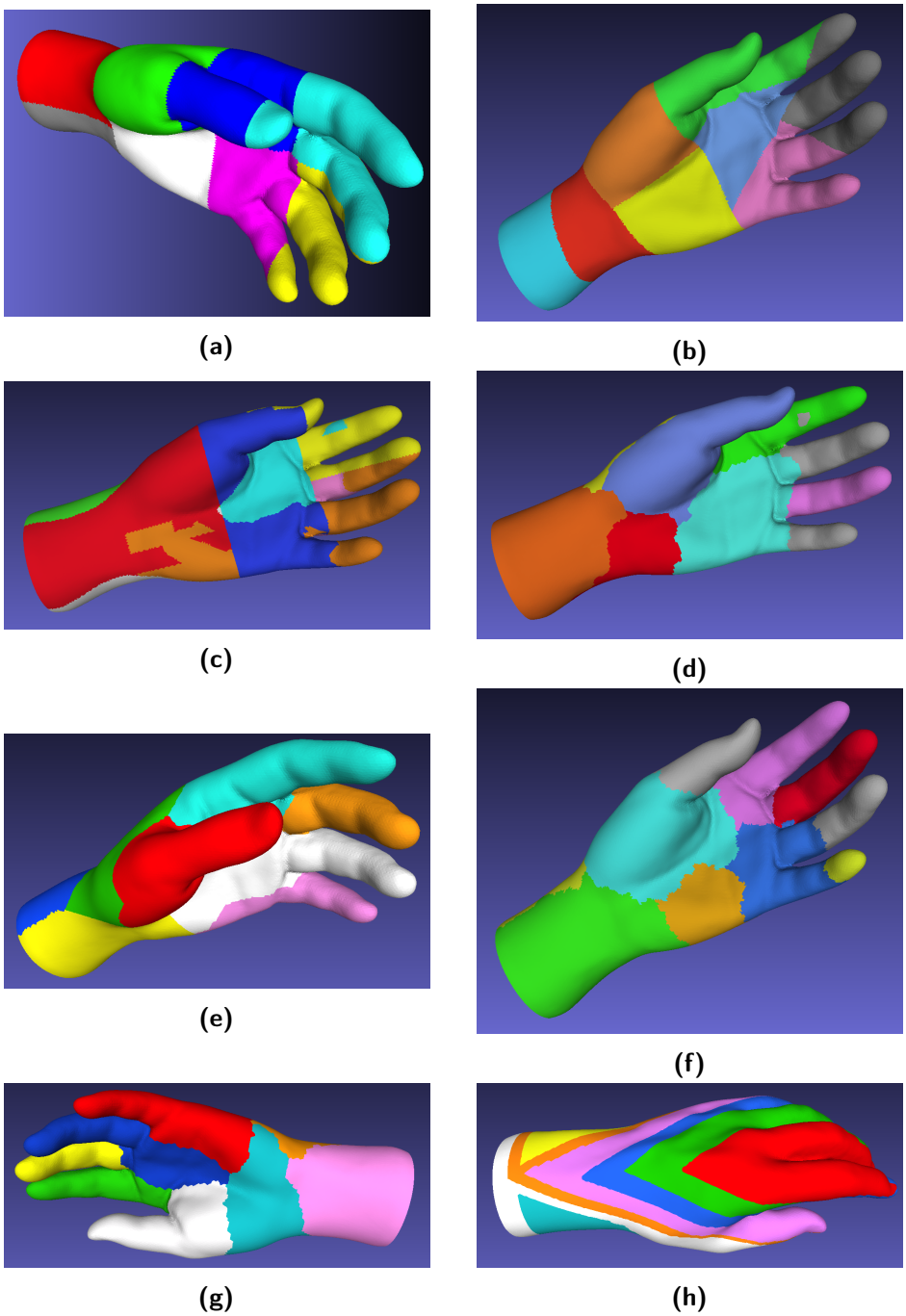


Figure 4.25: From top to bottom, left to right: Zoltan RCB, Zoltan RIB, Zoltan HSFC, Zoltan PHG, KaFFPa, METIS, ParMETIS and AD-tree partitioning on the mesh hand.

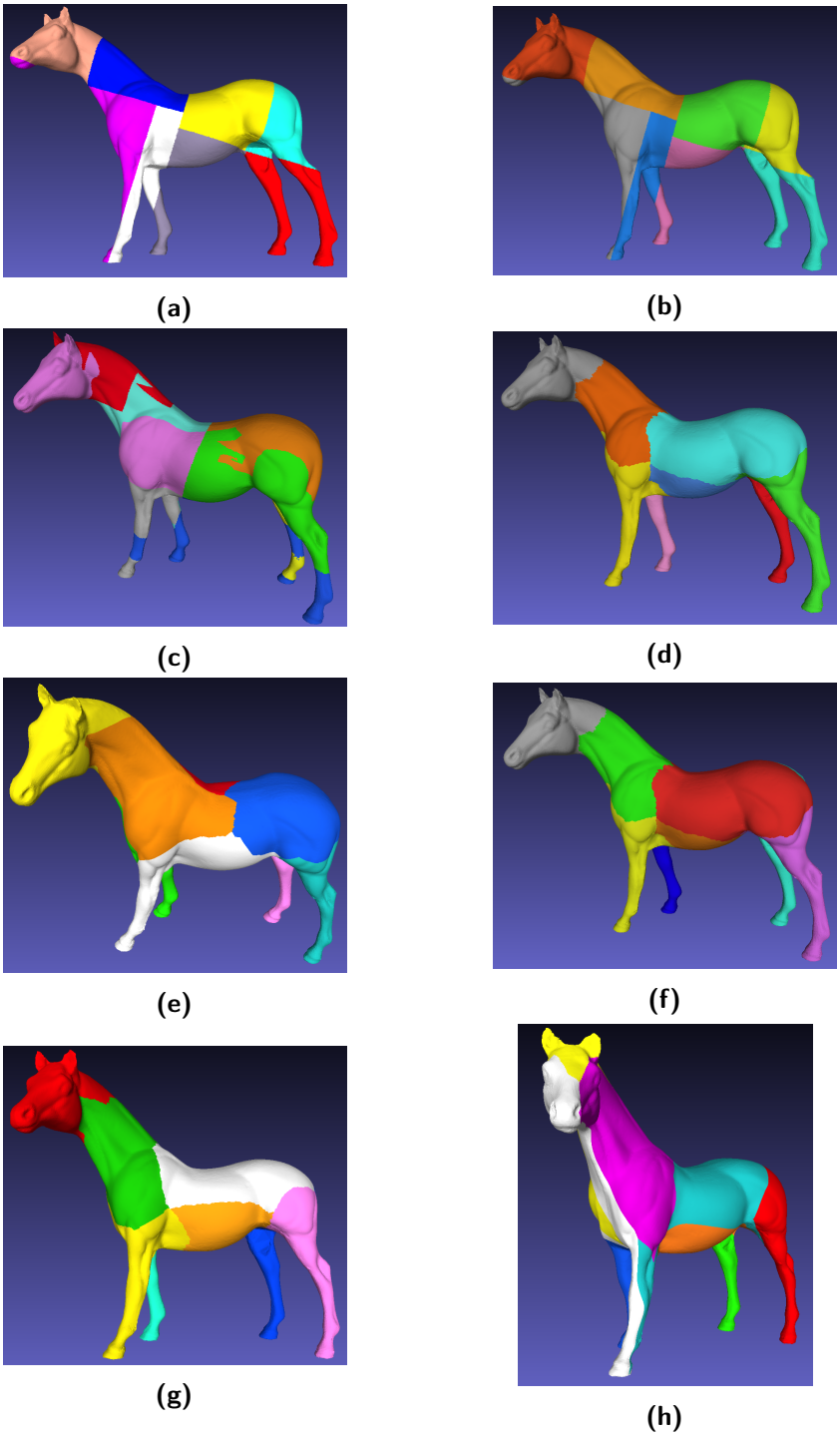


Figure 4.26: From top to bottom, left to right: Zoltan RCB, Zoltan RIB, Zoltan HSFC, Zoltan PHG, KaFFPa, METIS, ParMETIS and AD-tree partitioning on the mesh horse.

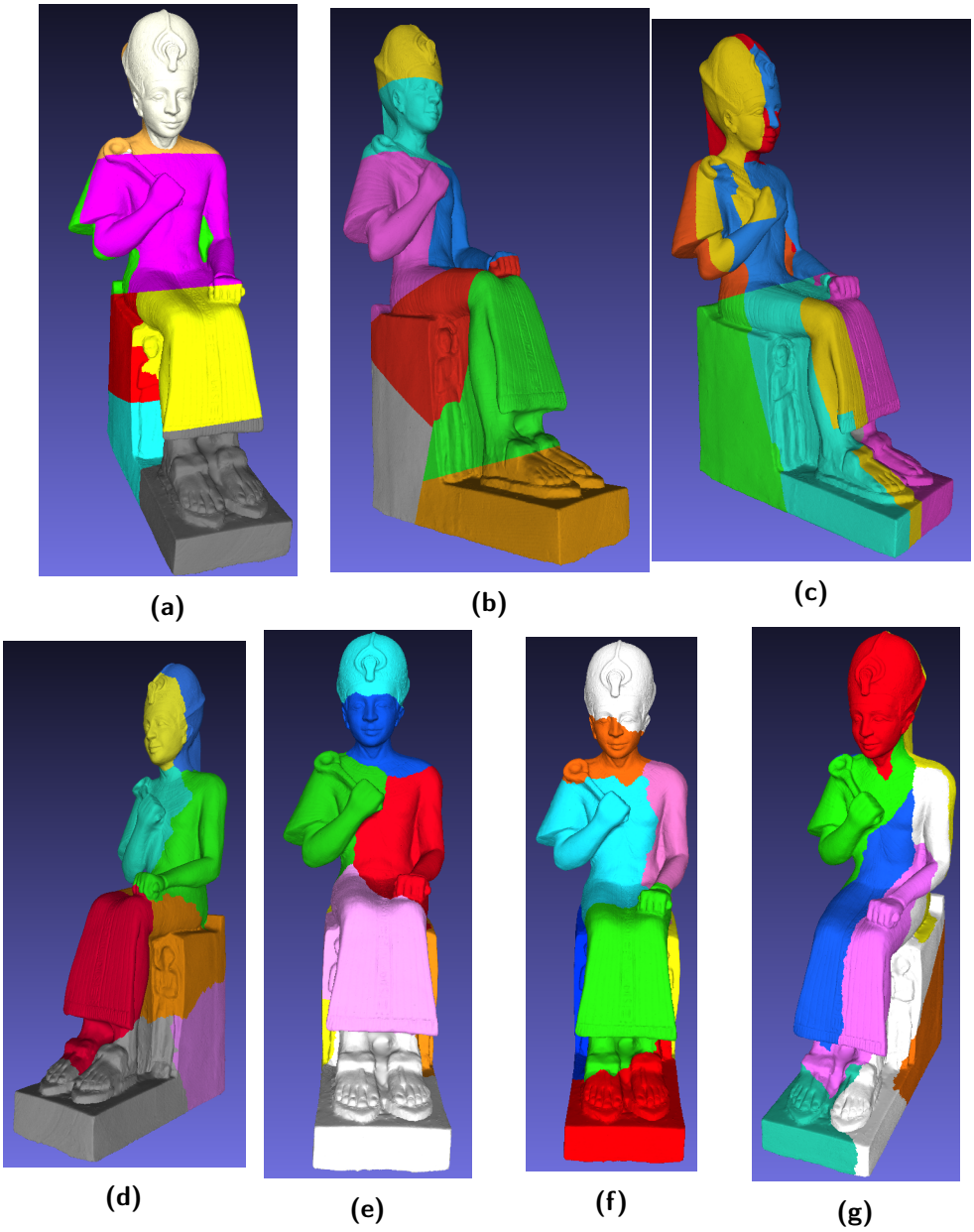


Figure 4.27: From top to bottom, left to right: Zoltan RCB, Zoltan RIB, Zoltan HSFC, Zoltan PHG, KaFFPa, ParMETIS and AD-tree partitioning on the mesh ramesses.

BUNNY	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	69666	3.926	1.000	1.000
Zoltan RIB		3.970	1.000	1.000
Zoltan HSFC		4.113	1.000	1.000
Zoltan PHG		4.603	1.021	0.832
KaFFPa		5.401	1.023	0.964
METIS		0.0953	1.470	0.686
ParMETIS		0.165	1.010	0.994
AD-tree part.		**0.0241 + 0.0126 (0.0367)**	1.118	0.799

Table 4.1: Comparison between partitioning algorithms on the mesh bunny.

CASTING	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	10224	3.819	1.000	1.000
Zoltan RIB		3.666	1.000	1.000
Zoltan HSFC		3.665	1.000	1.000
Zoltan PHG		3.654	1.041	0.967
KaFFPa		1.108	1.016	0.959
METIS		0.0165	2.353	0.524
ParMETIS		0.0411	1.029	0.971
AD-tree part.		**0.00254 + 0.000819 (0.003359)**	1.146	0.858

Table 4.2: Comparison between partitioning algorithms on the mesh casting.

COW	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	5804	3.701	1.000	1.000
Zoltan RIB		3.740	1.000	1.000
Zoltan HSFC		3.872	1.000	1.000
Zoltan PHG		3.778	1.062	0.982
KaFFPa		0.972	1.017	0.972
METIS		0.0142	1.522	0.566
ParMETIS		0.0306	1.028	0.957
AD-tree part.		**0.00276 + 0.000445 (0.003205)**	1.141	0.866

Table 4.3: Comparison between partitioning algorithms on the mesh cow.

DRAGON	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	100000	4.014	1.000	1.000
Zoltan RIB		3.943	1.000	1.000
Zoltan HSFC		3.956	1.000	1.000
Zoltan PHG		4.822	1.085	0.869
KaFFPa		5.621	1.026	0.934
METIS		0.188	1.080	0.922
ParMETIS		0.201	1.016	0.979
AD-tree part.		**0.0458 + 0.018 (0.0638)**	1.261	0.826

Table 4.4: Comparison between partitioning algorithms on the mesh dragon.

HAND	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	72958	3.998	1.000	1.000
Zoltan RIB		3.834	1.000	1.000
Zoltan HSFC		3.992	1.000	1.000
Zoltan PHG		4.723	1.090	0.969
KaFFPa		5.384	1.028	0.967
METIS		0.156	1.256	0.778
ParMETIS		0.169	1.021	0.978
AD-tree part.		**0.0259 + 0.0129 (0.0388)**	1.032	0.969

Table 4.5: Comparison between partitioning algorithms on the mesh hand.

HORSE	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	225280	4.346	1.000	1.000
textZoltan RIB		4.654	1.000	1.000
Zoltan HSFC		4.62	1.000	1.000
Zoltan PHG		6.868	1.077	0.981
KaFFPa		19.351	1.014	0.979
METIS		0.352	1.116	0.915
ParMETIS		0.381	1.002	0.998
AD-tree part.		**0.0464 + 0.0428 (0.0892)**	1.037	0.936

Table 4.6: Comparison between partitioning algorithms on the mesh horse.

RAMESSES	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	1652528	10.935	1.000	1.000
Zoltan RIB		10.905	1.000	1.000
Zoltan HSFC		11.824	1.000	1.000
Zoltan PHG		33.522	1.089	0.884
KaFFPa		200.848	1.029	0.919
METIS		n.c.	n.c.	n.c.
ParMETIS		2.883	1.001	0.999
AD-tree part.		**0.701 + 0.309 (1.010)**	1.147	0.786

Table 4.7: Comparison between partitioning algorithms on the mesh ramesses.

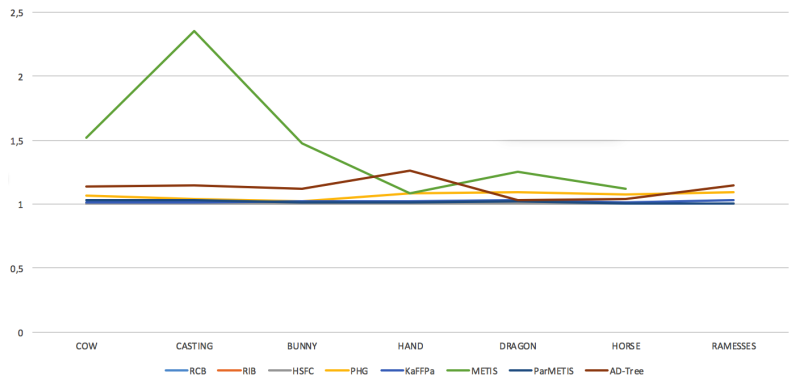


Figure 4.28: Graph representing the maximum imbalance of the partitioning algorithms on the considered meshes.

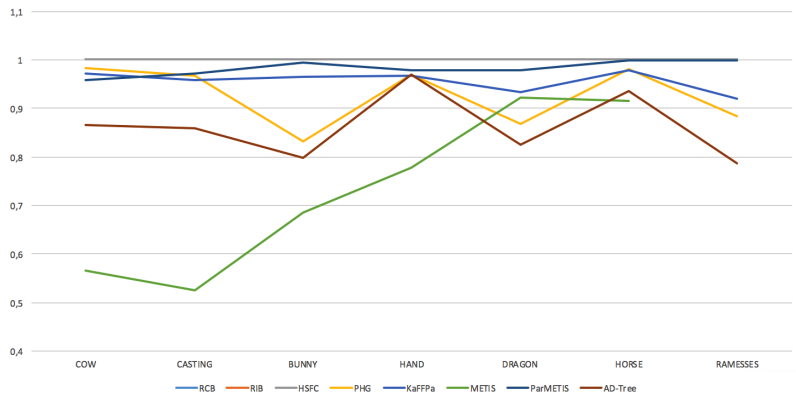


Figure 4.29: Graph representing the minimum imbalance of the partitioning algorithms on the considered meshes.

	RCB	RIB	HSFC	PHG
bunny	3.73	3.49	5.26	2.89
casting	10.4	9.9	13.8	8.33
cow	13	13	14.3	8.74
dragon	3.56	3.55	4.88	1.82
hand	3.56	3.56	5.4	3.41
horse	2.09	2.09	3.33	1.49
ramesses	0.72	0.71	1.32	0.579

Table 4.8: Average percentages of border nodes with respect to the partitions size, part 1.

	METIS	ParMETIS	KaFFPa	AD-Tree part.
bunny	3.01	2.75	2.39	4.31
casting	9.62	8.16	6.85	13.5
cow	15.63	7.51	7.22	11.3
dragon	1.72	1.45	1.24	4.35
hand	3.39	3.05	2.7	7.1
horse	2.73	1.48	1.33	2.25
ramesses	n.c.	0.5	0.462	1

Table 4.9: Average percentages of border nodes with respect to the partitions size, part 2.

One metric that has not been considered is the scalability with respect to the number of partitions. Differently from all the other algorithms, AD-tree partitioning is impervious to a change in k, as showed in Table 4.10 (considering, this time, only the partitioning time, since the tree creation does not change), for the mesh bunny.

The section is concluded with the partitioning obtained on the mesh derived as in [78–80], from the image sequence castle-P30 (Figures 4.30 and 4.31), belonging to the EPFL dataset [3]. The mesh obtained has around 4.1 million faces and is divided, with $k = 8$ in about 3 seconds (considering the conversion graph to AD-tree plus the partitioning).

Time	$k = 8$	$k = 16$	$k = 32$	$k = 64$
RCB	3.93	7.92	15.09	38.49
RIB	3.97	7.65	15.051	37.22
HSFC	4.11	7.54	14.62	28.26
PHG	4.60	10.69	18.77	37.32
KaFFPa	5.40	13.032	15.38	20.62
METIS	0.095	0.10	0.12	0.14
ParMETIS	0.165	0.17	0.176	0.185
AD-Tree part.	0.0126	0.0129	0.0131	0.0131

Table 4.10: Time difference with different k, for the mesh bunny.

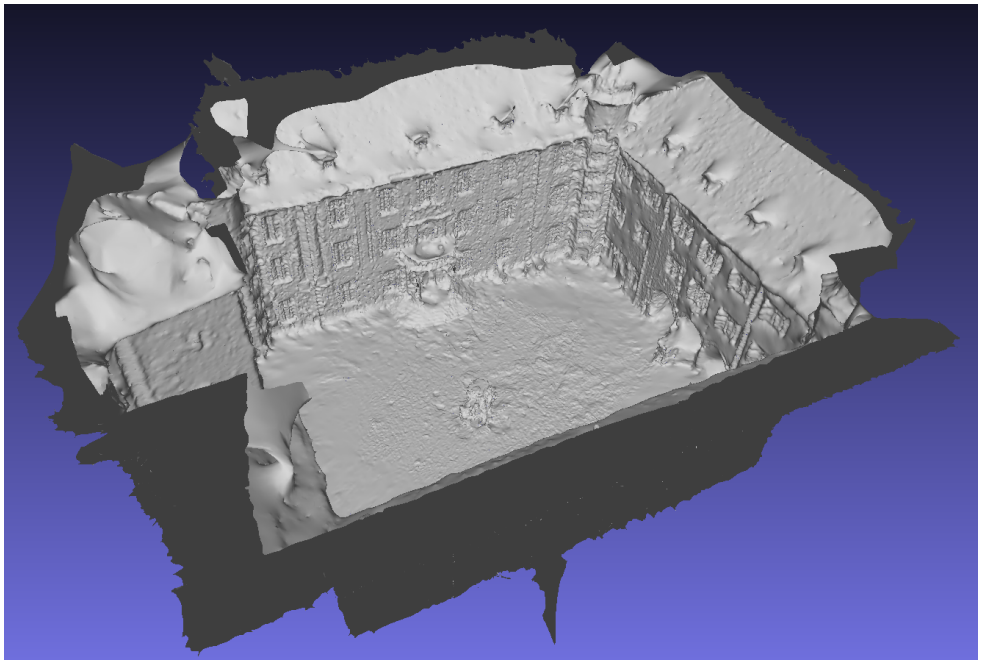


Figure 4.30: Mesh obtained from the castle-P30 image sequence.

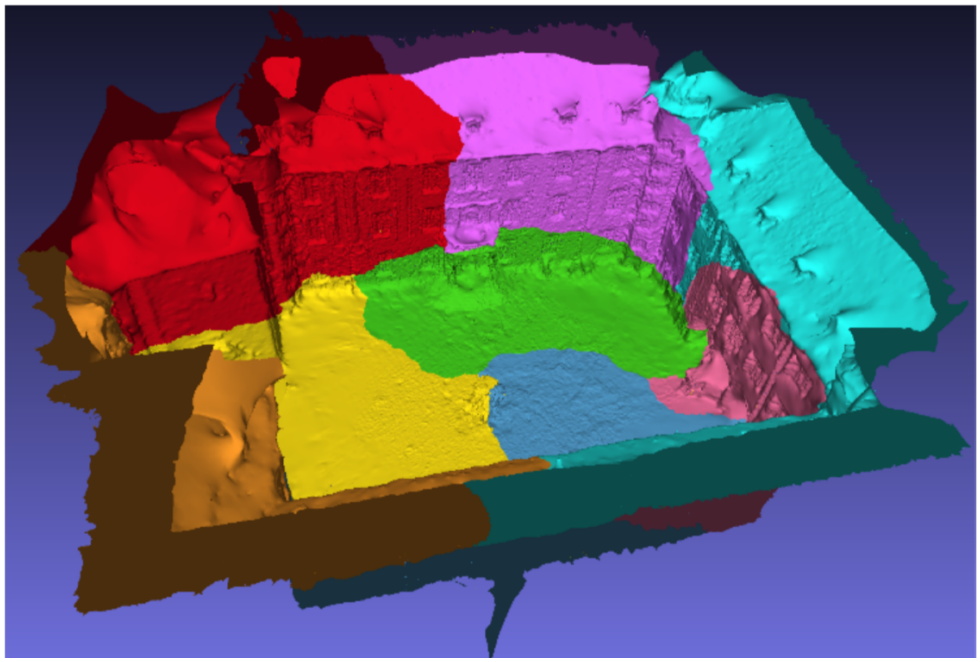


Figure 4.31: Mesh of Figure 4.30 partitioned into 8 parts.

5 DIRECTED PARTITIONING

In Chapters 3 and 4 we proposed an algorithm that exploits additional information of the graph, Label Clustering, and an algorithm that represent the graph with an augmented data structure. In this chapter we propose a partitioning algorithm that works directly on the graph, without needing additional data structures.

The idea behind the method is, once again, the depth-first algorithm, used to visit all the nodes of the tree. To partition a graph with $|V|$ nodes into k groups, the proposed method starts going through the nodes counting them, stopping when $|V|/k$ vertices are explored.

For example, given the graph in Figure 5.1a, a partition of size 5 can be the one represented in 5.1b. The partition is perfectly balanced, and depth first is used for this reason: as soon as the desired size is reached, the exploration stops. However, the obtained partition does not have fully connected nodes, but is characterised by a low amount of edges, so that the sum of weights inside the block is not maximised.

Another solution, more likeable, is instead the one represented in Figure 5.1c. The showed partition is compact and with a high amount of inner edges, and is the result that should be reached. Unfortunately, depth first does not discriminate between the two cases, since no spatial information is considered. In the next two sections, modifications to the depth-first algorithm to solve this problem are proposed. Then the partitioning algorithm, called Directed partitioning, is described, followed by experimental results. For all the rest of the chapter, it is assumed that the adjacency list of a node expresses neighbours ordered with a counter-clockwise sense, and not randomly. For example, one possible adjacency set for node 0 in Figure 5.2 is given by the sequence {1,3,5,2,4}.

5.1 LOCAL (EXPLORATION) STRATEGY

Looking at Figure 5.3, one can easily understand how a node should be selected for the exploration. The initial seed of the partition, painted in red, explores all of its adjacent nodes and selects one randomly, the orange one. This node is now explored, having all the neighbours visited.

To obtain the desired partition, assuming the exploration counter-clockwise, the green node must be selected as the next element to be explored, but the adjacency list of the orange node does not give any indication about the green node position. However, since the adjacency list is ordered, it is known

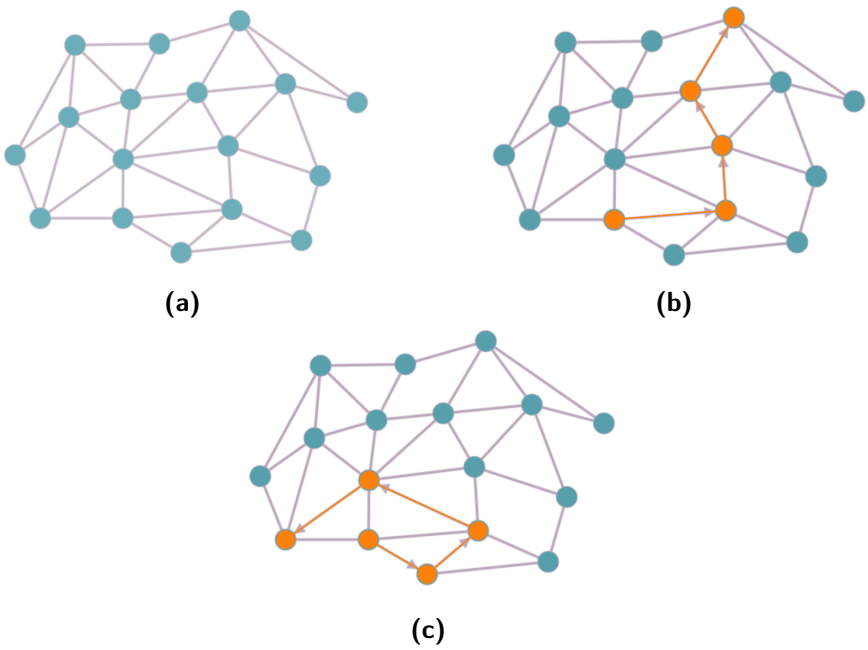


Figure 5.1: If pure depth-first is applied on the graph to the left, any kind of partition could be obtained, like Figure 5.1b, while instead a more patch-like block is desired, as in Figure 5.1c.

that the green node is right behind the red one in the list (because it is the first clockwise). In the same way, the green node is the most distant from the red node in the list, going circularly left to right. The idea is then to shift the elements of the adjacency list such that the first element in the list is the node from which the current node comes from (in the example, we want the red node to be the first in the adjacency list of the orange node). The shifting operation can be done because it does not change the order of the elements in the list. In this way, it is known both which node should be explored next, i.e. the last of the list, and the order in which all the other neighbours should be explored later.

It is obvious that going clockwise inverts almost all of what as been said until now. While the adjacency list has the same meaning, the first node to be explored will be the second of the shifted list (not the last, considered before).

Let's see with an example how this procedure, called *local strategy*, works, considering the graph in Figure 5.4a. Node 1 is the seed that starts the exploration, and visits its adjacent nodes, say that they are ordered as {0,5,6,2,3} (Figure 5.4b). Since it is the first node explored, the next node is selected randomly, suppose node 2 (Figure 5.4c). Let the neighbours of it be ordered as {12,9,3,0,1}. Following the local strategy, we shift the adjacency set until the previous node is first in the list: {1,12,9,3,0}. In this way we know that

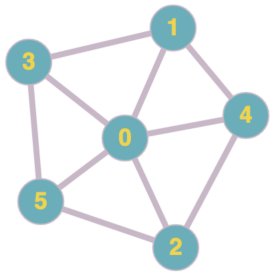


Figure 5.2: Simple graph to demonstrate how the adjacency set considers nodes.

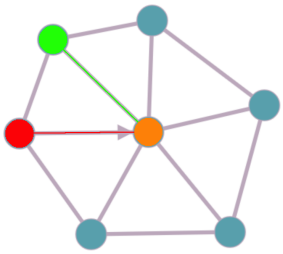


Figure 5.3: Representation of the ideal order of exploration: red, orange and then green nodes.

the next node to be explored is the last of the list, 0 (Figure 5.4d). The same reasoning can be applied for this node: supposed that its adjacency list is {6,1,2,3,7,4,8,5}, it is shifted to obtain the sequence {2,3,7,4,8,5,6,1}. Node 1 is already explored, so the algorithm selects the last but one from the end of the shifted adjacency list (Figure 5.4e). The local strategy is then applied for all the other nodes in the same way.

5.2 GLOBAL (EXPLORATION) STRATEGY

It has been seen how each node selects the next element to be explored, shifting its adjacency list to understand the relative position of the neighbours. In this section four global strategies representing how the exploration proceeds in the whole graph are described. The strategies are applied on the graph associated to the faces of an icosahedron, represented in Figure 5.5.

5.2.1 *Depth first with no visiting*

When, using depth-first, a node goes through its neighbours, it marks all of them as visited and selects one to explore, using the local strategy described before. In this modification of the algorithm, instead of being marked as visited, nodes are immediately explored.

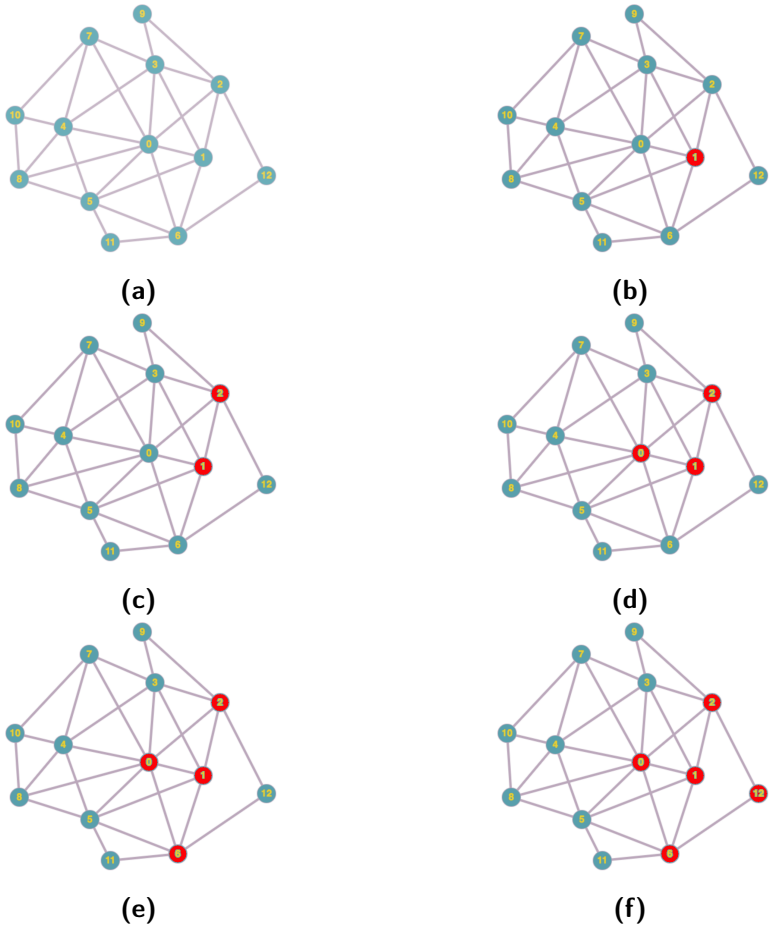


Figure 5.4: Example of how the local strategy works.

Consider the graph represented in Figure 5.5c, starting from node f_0. The node marks all of its neighbours as explored, and selects one randomly to continue the algorithm, let's say f_5, whose free adjacent nodes are f_3 and f_6. Both are marked as explored, but only the latter is used to continue. This procedure goes on until the desired number of nodes is explored (considering all the marked nodes, and not only the ones used to go around the graph). Figure 5.6 shows the whole exploration of the considered graph.

The main advantage of this algorithm is the speed and low memory usage: nodes are immediately marked as explored so that there is no need to insert them into a stack (as in pure depth first). However, frequent holes can be created in the graph, as showed in Figure 5.7, and the obtained partition can be elongated in shape.

The hole generation problem heavily depends on the graph structure and happens when all the nodes around a subset are explored before reaching its elements during the algorithm. The second problem is related to the explo-

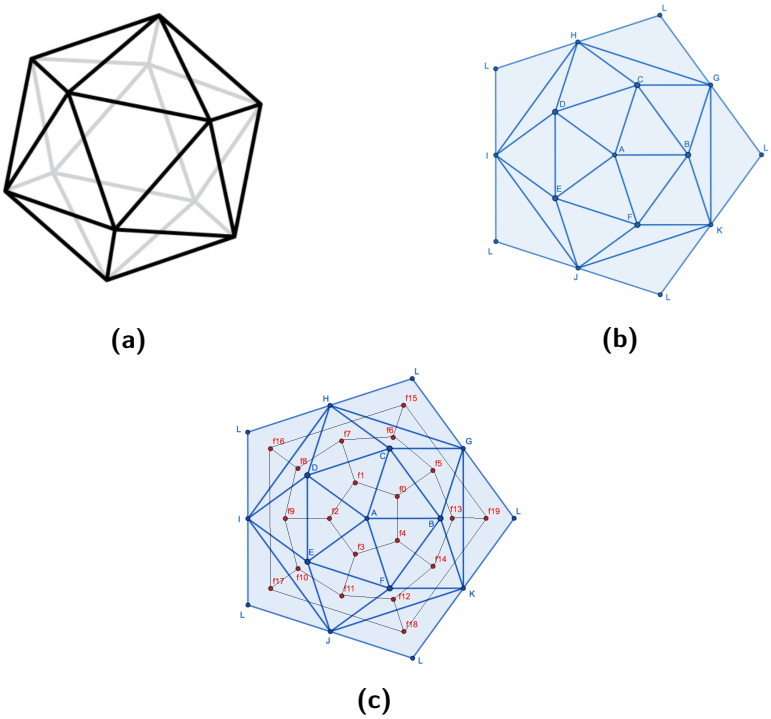


Figure 5.5: Icosahedron used to explain different global strategies. The graph representing its faces, to the right (Figure 5.5c), will be the on which such strategies are applied.

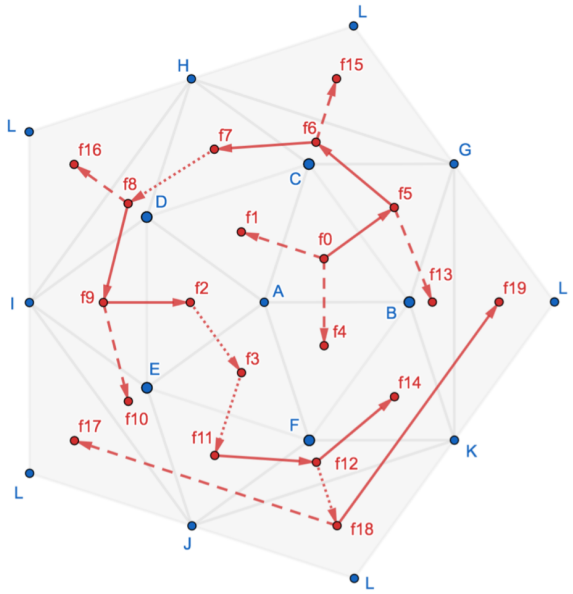


Figure 5.6: Depth first with no visiting on the icosahedron's graph.

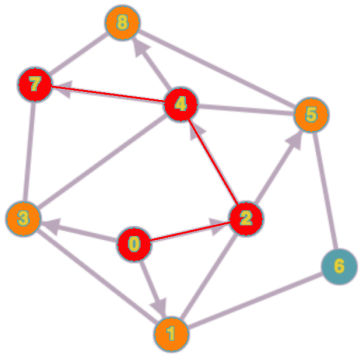


Figure 5.7: Depth first with no visiting can create multiple holes in the graph, as node 6. Red nodes are explored while orange nodes are marked as explored.

ration order in the graph, partially enforced by the fact that for each node, all the neighbours are explored and only one continues the exploration.

5.2.2 *Depth first with alternated expansion sense*

Let v be the node being currently explored, whose adjacent nodes are $\{a_0, a_1, \ldots, a_n\}$. In pure depth first, if a_n is already explored, the next node in the algorithm would be a_{n-1}. A variant of the procedure handles differently the nodes already explored. If up to the moment v is reached, nodes were traversed counter-clockwise, the sense changes to clockwise, and the first explored node will be a_1. Vice versa, if the traversing sense was clockwise, it changes to counter-clockwise and the next node to be explored will be a_{n-1}.

An example is showed in Figure 5.8. Starting from node f_0, the exploration starts counter-clockwise, until node f_1 is reached. The candidate to be explored is f_0, that is already explored. The new traversing sense changes to clockwise and the next node to be considered is f_2.

The main advantage of this algorithm is that the partitions created have regular shapes, patch-like or with flowers. However, it creates frequent holes, as it can be seen in the example.

5.2.3 *Depth first and last*

Moving only counter-clockwise, nodes are visited and expanded as the normal depth-first algorithm describes. A queue is used to support the algorithm. The front element in the queue is the current explored node, and each time a node is visited (but not explored), it is inserted in the queue. When a node is explored, it replaces the front element of the queue. When the algorithm encounters a node that is already explored, the front element

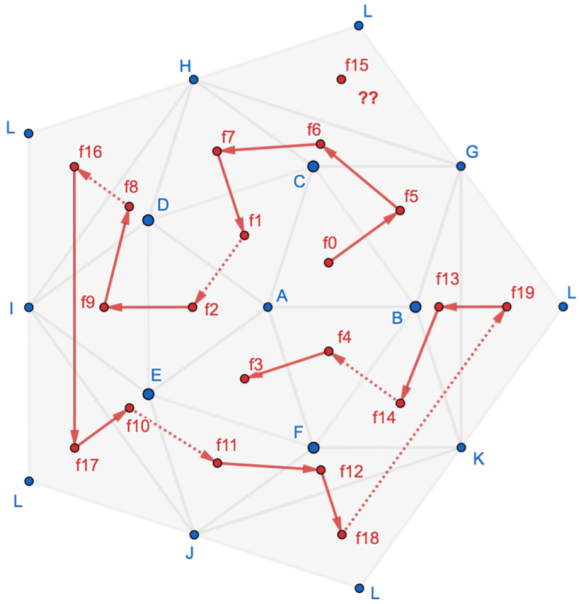


Figure 5.8: Depth first with alternated expansion sense on the icosahedron's graph.

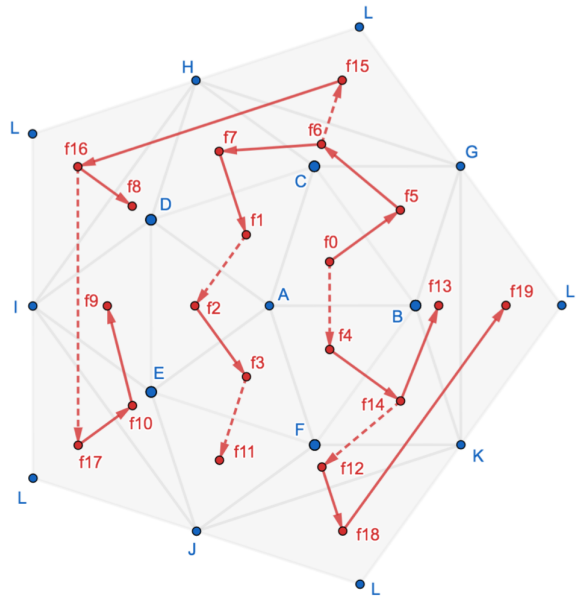


Figure 5.9: Depth first and last on the icosahedron graph.

of the queue is removed instead of being replaced. The new front element is the next node that will be explored by the algorithm.

An example is given by Figures 5.9 and 5.10. The seed starting the algorithm is node f_0, inserted in the queue (Figure 5.10a). The node visits its neighbours, and selects f_5 to be expanded, replacing it into the front spot in the queue and inserting the other visited nodes (Figure 5.10b). f_5 has two

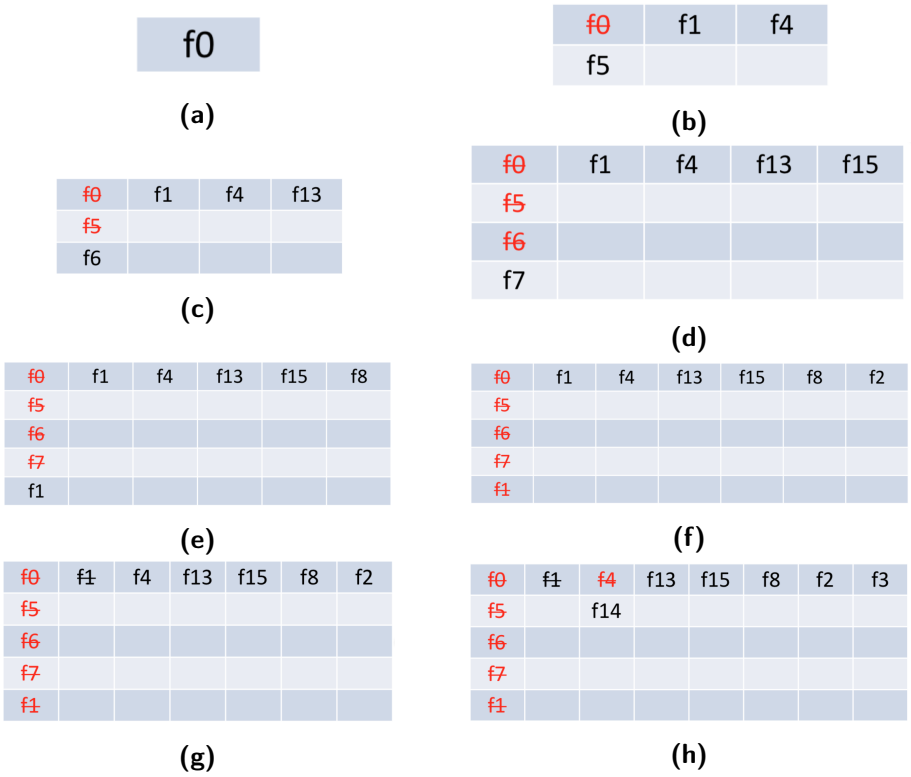


Figure 5.10: Status of the queue in the first iterations of the depth first and last algorithm used on the icosahedron's graph.

adjacent nodes: f_6 replaces it and f_{13} is inserted in the queue (Figure 5.10c). The same is done for node f_6, replaced by f_7 and allowing the insertion of f_{15} in the queue (Figure 5.10d). The algorithm continues exploring f_7, that selects f_1 as next node, putting f_8 at the end of the queue (Figure 5.10e). However, the candidate node for the next expansion, f_0, is already inserted. The front of the queue is popped after having added the visited nodes (Figure 5.10f) and the new front is considered the node to be explored. Since it is already part of the current group, the queue front is once again popped (Figure 5.10g), leading to the new explored node f_4. The same steps are repeated: the visited node, f_3, is added to the queue and the explored node is replaced with the new candidate f_{14}. These procedure continues until the desired number of explored nodes is reached.

It is interesting to notice that a closure, meaning the action that would explore a node already considered, is detectable in the queue even before checking the neighbours of a node. If the front element is also located in another position of the queue, a closure is found, meaning that a full counter-clockwise path has been traversed.

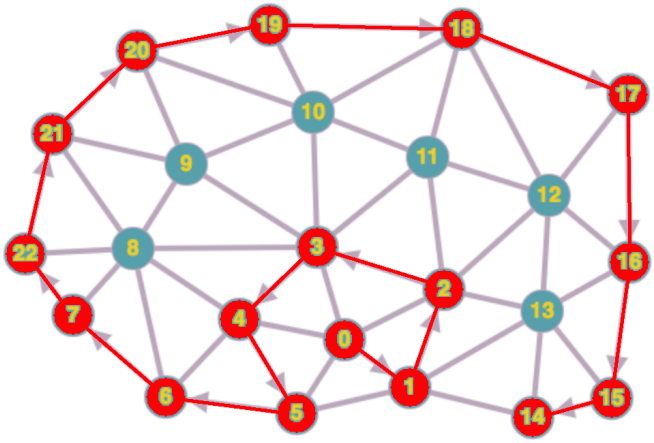


Figure 5.11: Ring-like partition created with pure depth first.

While the algorithm is easy to implement and has some interesting properties, like the pre-emptive closure detection just described, it can still create holes in the graph, and the shape of the obtained partition can be irregular.

5.2.4 *Depth first with border detection*

This algorithm is particularly suited for graphs whose nodes have mostly the same degree, like graphs derived from meshes. It works the same way as depth-first, starting to expand nodes counter-clockwise. Each time a border of the graph is encountered, meaning a node with degree less than the maximum one in the graph, the sense of exploration is inverted, and the algorithm proceeds in the usual way, without other modifications. This shrewdness is needed to avoid partitions that show ring-like structures, as in Figure 5.11. This variant of depth first is the one used in the partitioning algorithm as global strategy, for two reasons. First, it generates patch-like blocks, exploring nodes attached to the subset of elements already explored. Second, it generates a limited amount of holes, for the same reason.

5.3 HOLES AND HOW TO REMOVE THEM

All the algorithms described in the global strategy section presented a major drawback: they generate holes. Holes are visual artefacts that cannot be avoided, since they depend on the structure of the graph, often having convoluted and intricate connections. Giving a proper definition of hole is difficult, because one should consider ambiguous situations, like in Figure 5.12.

Anticipating how the partitioning algorithm works, if a graph has to be divided into k blocks, a viable solution is to bisect it recursively, halving the

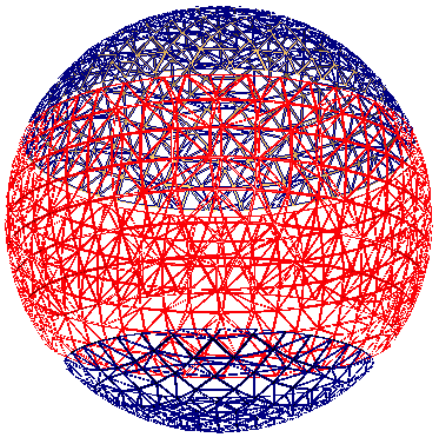


Figure 5.12: Given the partition in red, which of the two blue sub-graphs can be considered hole?

size at each iteration. The bisection of a graph, using the global and local strategies described before, induces three components:

- The *main partition*, obtained through the visiting algorithms and seen strategies
- The *other-partition*, that is the set of vertices that do not belong to the main partition
- Holes, that are sets of nodes that should belong to the main partition, but are separated from both the main partition and other-partition

While distinguishing the first component is an easy task, since it is determined univocally by the exploration procedure, currently there is no way to distinguish the other-partition from the holes. Consider again Figure 5.12, that shows the main partition in red. One cannot say if the hole is the bottom set of blue nodes or the top one. A convention is needed, described with new concepts.

The first entity that is necessary to distinguish holes from the other-partition is the border of the main partition, that visually is the set of nodes surrounding it.

Definition 5.1. Border The border of the main partition is the set of nodes obtained through the following steps.

1. The last and the second to last explored nodes, v_{last} and v_{last-1} are considered respectively the first and second elements of the border.

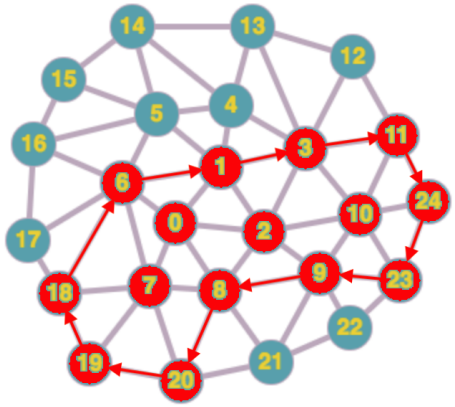


Figure 5.13: Given the partition in red and the last two inserted nodes 6 and 1, the border is detected going counter-clockwise from 1, considering 6 as its originator node.

2. Using the local strategy, the main partition is traversed again, with pure depth first and starting from v_{last-1} going only counter-clockwise, considering v_{last} as the node from which it came from. This means traversing the nodes of the partition in alternated direction with respect to the main exploration (if the last nodes visited are, in order, 1, 2 and 3, the new traversal explores 3, 2 and 1), to obtain a ring-like structure. This is done until v_{last} or an already explored node is encountered, meaning that the algorithm looped through the utmost nodes of the partition.

An example is given in Figure 5.13, where the border of the red partition (made unbalanced to explain the example) is detected starting from nodes 6 and 1, and is indicated with red arrows.

The border of the main partition defines an ordered sequence of nodes. As seen before, each node of the graph belongs to a certain group: the main partition, the other-partition or the holes. Having defined the border of the main partition, it is easy to see how the other-partition can be detected.

For each element of the ordered sequence of nodes that defines the border, its neighbours are computed. The first neighbour found that does not belong to the partition is the initial seed of the other-partition. Applying any visiting algorithm, like depth-first or breadth-first, from the initial seed and with the condition of exploring only nodes not belonging to the main partition, a set of vertices is obtained: this is the other-partition.

Continuing with the example started in Figure 5.13, the following step is to go through the ordered sequence of nodes forming the border. Its first element, 6, has seven neighbours. At least one of them does not belong to the main partition, say node 16, and such node is entitled as initial seed of the other-partition (5.14a). Then, a visiting algorithm is applied, suppose

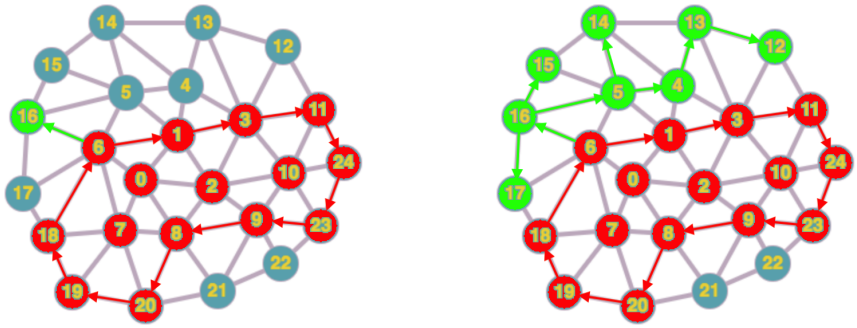


Figure 5.14: Detection of the initial seed and other-partition for the graph of Figure 5.13.

breadth-first, starting from the initial seed (node 16). The set of nodes explored in this way determines the other-partition, showed in Figure 5.14b. Two of the three components of graph bisected through visiting methods have been found, namely the main partition and the other-partition. By exclusion, the remaining one can be now defined.

Definition 5.2. Hole A hole is a set of nodes in a graph such that they do not belong to the main partition or the other-partition obtained through bisection, following local and global strategies. These nodes and are surrounded by nodes belonging to the main partition.

The number of nodes $|V|$ to the size of each component are related in the following way:

$$|V| = |\text{main partition}| + |\text{other-partition}| + \sum_i |\text{hole}_i|$$

Let notice that there is no constraint over the size of the other-partition and the size of the holes. It may happen that the other-partition is incredibly small with respect to the size of the holes. This is a problem because holes are added to the main partition (as in Figure 5.15), increasing its size and straying from the balanced property that was originally respected.

5.4 UNRAVELLING

When "filling the holes" of the main partition, its size increases, often dramatically, destroying the balanced property. To regain it, a process similar to osmosis can be applied between the main partition and the other-partition. If solvent molecules go through a selectively permeable membrane into a region of higher solute concentration, in the direction that tends to equalize the solute concentrations on the two sides, in the same way nodes migrate

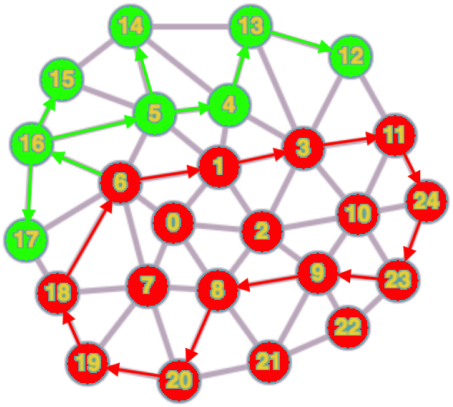


Figure 5.15: The hole detected in 5.14b is filled and assigned to the main partition.

from the zone with greater size to the smaller one, to balance both partitions. This migration is not random, as it happens for the solvent molecules, but follows a defined criterion.

Let v be the last node explored in the main partition. From v, the main partition is visited again, in a way similar to the one described when detecting its border: moving on the partition contour. Each explored node is transferred to the other-partition, until the main partition is once again balanced. This procedure is named unravelling since it removes a sequence of nodes as if they where a single long thread, detached from the rest of the hank that is the main partition.

Differently from border detection, unravelling must not go around the main partition contour, since this would produce ring-like structures in the other-partition. For this reason, when reaching the graph border (not to be confused with the partition border), the exploration sense is inverted (from counter-clockwise to clockwise and vice versa).

The example started in the last section, in Figure 5.13, can now be concluded. The graph has twenty five nodes and the main partition, now augmented with nodes that belonged to holes, has 17 elements. Starting from node 6, immediately assigned to the other-partition, nodes are unravelled in the following order: 1, 3, 11 and 24, all migrated to the other-partition. In particular, when reaching node 11, that is a node belonging to the graph border, the sense of exploration is inverted from counter-clockwise to clockwise (meaning that if the procedure continued, the next node to be unravelled would be 10 and not 23). In this way, two perfectly balanced partitions have been obtained, represented in green and red in Figure 5.16.

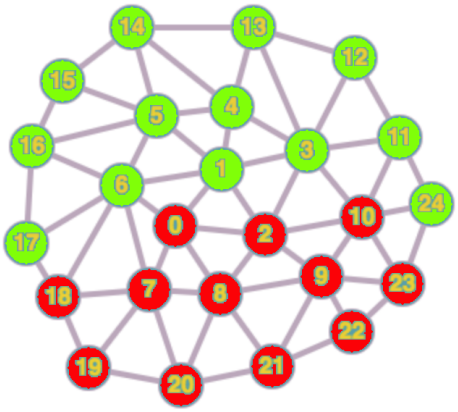


Figure 5.16: Partitions obtained after the unravelling procedure.

5.5 DIRECTED PARTITIONING

Local and global exploration strategies, holes detection, filling and nodes unravelling form, together, the steps of the proposed Directed partitioning algorithm. In this section, the method is explained avoiding repetitions, followed, as usual, by an example on a small graph.

5.5.1 *Partitioning algorithm*

In the previous sections problems and solutions regarding the partitioning of a graph, using visiting algorithms, have been seen. All the presented concepts can be grouped in the main partitioning algorithm, called directed partitioning. The idea behind direct partitioning is recursive bisection. To obtain k partitions, the graph is first divided into two blocks. Each partition is then broken again into two other parts, and so on.

The core of the algorithm is the bisection procedure described in Algorithm 13. Starting from an initial node, following the local depth-first with border detection strategies, the first bisection of the graph is computed, exploring a certain amount of nodes (starting at line 3, Algorithm 13). Then, the border of such partition is detected and the initial seed for the other-partition is selected (starting at line 9, Algorithm 13). Next, the initial seed is used to explore all the nodes that do not belong to the main partition, assigning them to the other-partition (starting at line 12, Algorithm 13). Having both the main partition and the other-partition, the remaining nodes of the graph that are not yet assigned to any group represent the main partition holes. They are assigned to it, filling the partition (starting at line 24, Algorithm 13). This increases the size of the main block, that has to be reduced through the osmosis/unravelling procedure, removing nodes from it

Algorithm 13 Directed Bisection

```
1: procedure DirectedBisection(G,initV,cutThresh)
2:     number of explored nodes numEN ← 1

3:     mark initV as the first explored node
4:     while numEN ⩽ cutThresh do
5:         explore a new node cn on the graph, following the local and
   depth first with border detection global strategies
6:         assign the node to the main partition, cn.group ← 0
7:         numEN ← numEN + 1
8:     end while

9:     let ln be the last node explored on the graph and assigned to the
   main partition
10:    detect the set of nodes B that form the main partition's border, start-
   ing from ln
11:    detect the initial seed is of the other-partition, using the nodes in B

12:    initialise the stack of the other-partition's nodes OPN
13:    insert is into OPN
14:    while OPN is not empty do
15:        current node cn ← OPN.top
16:        remove the top of OPN
17:        for all adjacent node cn_adj of cn do
18:            if cn_adj.group = −1 then
19:                cn_adj.group ← 2
20:                push cn_adj on OPN
21:            end if
22:        end for
23:    end while

24:    insert all the nodes that do not belong to group 1 or 2 into the set
   HOLES
25:    for all v ∈ HOLES do
26:        v.group ← 1
27:    end for

28:    let sizeMain be the number of nodes with group field equal to 1
29:    starting from ln, unravel nodes (assigning them to group 2) until
   sizeMain < cutThresh
30: end procedure
```

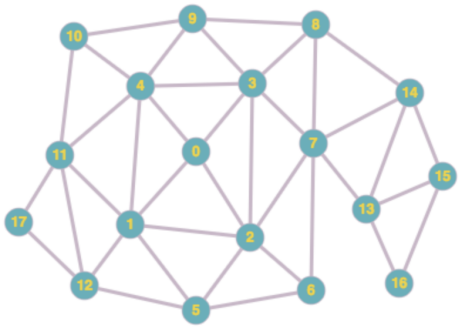


Figure 5.17: Graph used in the bisection example.

and assigning them to the other-partition (starting at line 28, Algorithm 13), until the desired dimension is reached.

5.5.2 *Partitioning example*

The section ends with an example of the bisection procedure applied on the graph showed in Figure 5.17. The following convention is used: red nodes are explored, orange nodes are visited and green nodes belong to the other-partition.

The algorithm starts considering node 7 as the first explored node, having neighbours {2,6,13,14,8,3} (Figure 5.18a). Remember that it is assumed that the adjacency list is ordered counter-clockwise. The next node is selected randomly, 14, and visits nodes 13, 15 and 8 (Figure 5.18b). Following the local strategy, the successive node to be explored is 8, because it is the last node considering counter-clockwise the adjacent nodes of 14, starting from its originator 7. Node 8 has only two unexplored adjacent nodes to visit: 3 and 9 (Figure 5.18c). Again adopting the local strategy, the next explored node is 3, having neighbours {9,4,0} (Figure 5.18d), followed by node 2 with neighbours {1,5,6} (Figure 5.18e).

The next explored node is 6 (Figure 5.18f), that has only 5 as unexplored neighbour. 6 is the first node of the graph border whose desired neighbour, 7, is already explored: the exploration sense changes from counter-clockwise to clockwise. 5 is the next node, with two neighbours: 1 and 12 (Figure 5.19a). The exploration sense is now clockwise, so next node to be considered during the exploration is 1 (instead of 12, that would erroneously continue the algorithm selecting the border nodes, leading to a ringed partition). Its adjacent nodes are {0,4,11,12} (Figure 5.19b). Still going clockwise, the next node to be explored is 0, that is also the last since nine nodes have already been explored (half of the number of vertices in the graph, Figure 5.19c).

The first phase of the bisection algorithm is concluded, and now the partition border must be detected. Since the graph is small, the border coin-

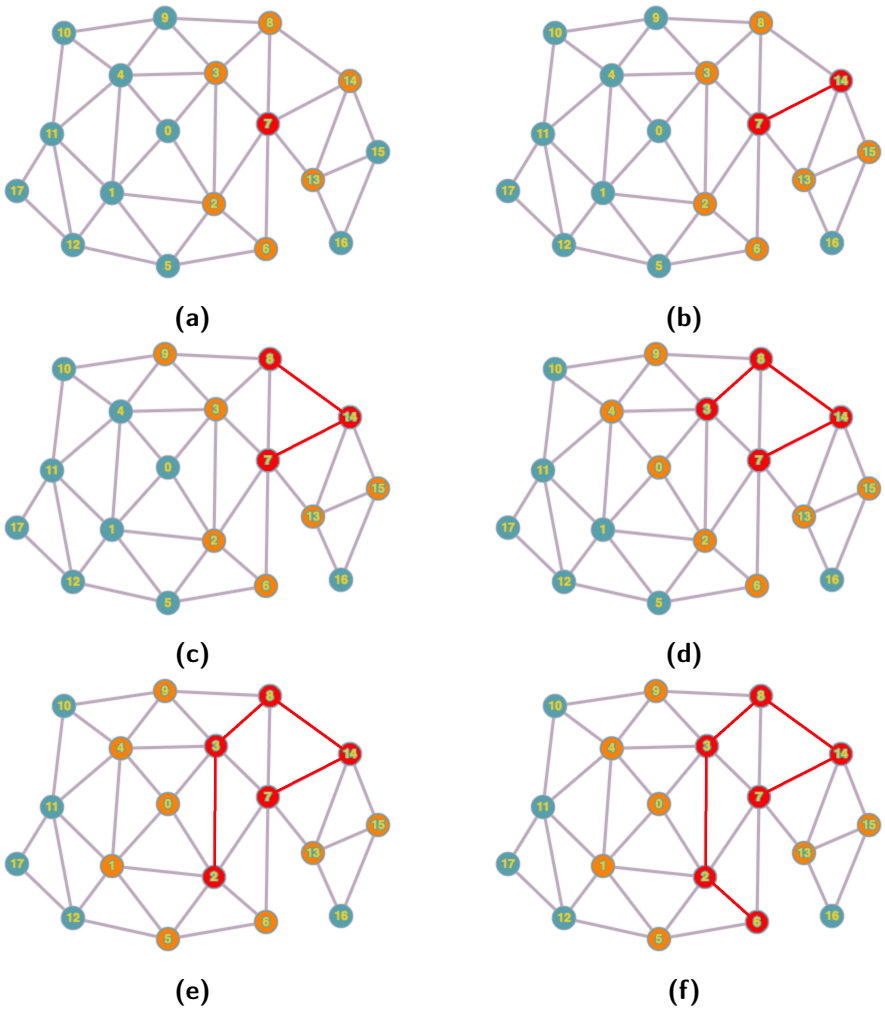


Figure 5.18: First part of the graph bisection example.

cides with the partition itself. Starting from the last node inserted, 0, the first neighbour that is not member of the main partition is 4, marked as the initial seed of the other-partition (Figure 5.19d). Starting from it, graph is explored using pure depth first, not considering the nodes belonging to the main partition. In this way, the other-partition is obtained (Figure 5.19e), containing six nodes.

A hole is detected, formed by nodes {13,15,16}, and is added to the main partition (Figure 5.19f), that now has twelve elements, more than the desired size. The last action to do is unravelling. Starting again from the last explored node, 0, the main partition is traversed, and each node passed is assigned to the other-partition. Nodes 0, 1 and 5 are detached from the main partition, that reaches the desired size of nine elements (Figure 5.19g). The bisection is now complete and results in perfectly balanced partitions.

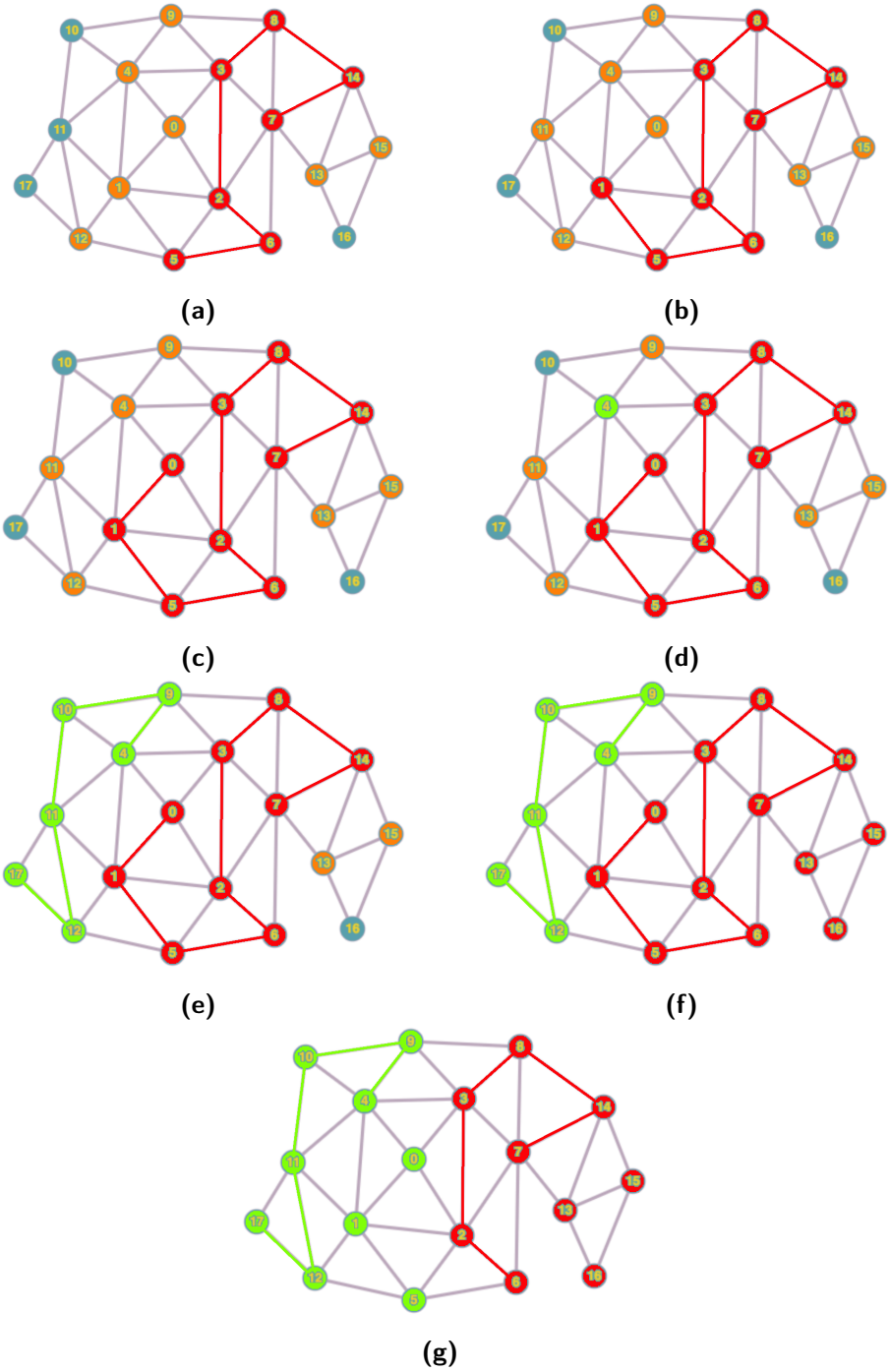


Figure 5.19: Second part of the graph bisection example.

5.6 DIRECTED PARTITIONING RESULTS

To demonstrate the results of the Directed partitioning algorithm, we use the graphs derived from the mesh collection provided with the benchmark described in [4] (considering their faces as nodes), comparing them with the main competitors in terms of perfect balance of the partitions: Zoltan RCB [75] and ParMETIS [64, 65, 63, 60], for $k = 4$.

The following configuration is used.

- Values are taken as average over different runs on a 64-bit laptop with Intel® Core(TM) i5-3337U CPU @ 1.80GHz x 4 processors, each with 3072Kb of cache size.
- Zoltan RCB is set to work in parallel using as many threads as k.
- ParMETIS uses the *ParMETIS_V3_PartMeshKway* routine to compute the partitions directly on the surface mesh. Default parameter values are used, as suggested in the toolkit manual [60]. Only one thread is considered.
- Directed partitioning is applied three times sequentially. With the first pass, a mesh is bisected into two parts. Then, each partition is bisected into other two blocks, forming four partitions in total.
- The first pass of directed partitioning starts on a random node of the mesh, while the others begin from a random node of the partitions borders, providing a better result.
- Multiple instances of the partitioning algorithm are run in parallel. The one that ends first (providing also the best result) is considered, discarding the remaining ones.

To evaluate the three algorithms, the same metrics adopted in Chapter 4 are used:

- Execution time, computed for Directed partitioning as the sum of each pass
- Maximum and minimum imbalance, useful to judge the ability of a procedure to obtain balanced partitions; they are respectively computed as

$$\frac{\max |partition_i|}{|\text{perfectly balanced partition}|}$$

and

$$\frac{\min |partition_i|}{|\text{perfectly balanced partition}|}$$

- The percentage of border nodes of a partition with respect to its size, defined as
$$\frac{\text{num border nodes of } partition_i}{|\,partition_i\,|}$$

Lower values of the three metrics correspond high quality partitioning.

The first contribution of the proposed algorithm is that it works directly on the data structure, be it meshes rather than graphs, without the need of a conversion of an auxiliary data structure.

Figures 5.20, 5.21, 5.22, 5.23, 5.24, 5.25, 5.26, 5.27, 5.28 and 5.29 represent the resulting partitions of the considered meshes, showing the outcome of each algorithm. As it can be seen by the figures, to RCB corresponds hard partitions, meaning that visually they are rectangular blocks (motivated by the fact that these methods partition the space and not the mesh itself). ParMETIS and Directed partitioning produce, instead, smooth and enveloping partitions.

Let's consider now the metrics associated to the meshes considered, described in Tables 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9 and 5.10. Directed partitioning excels over all the other algorithms in terms of execution time. If all the partitions are executed serially, as considered in the tables, the complexity is linear with $\log_2 k$, meaning that with more partitions we want to obtain, the computational time increase. To mitigate this problem, each partition can be assigned to a thread and executed in parallel, changing the bottleneck to the slowest thread.

Considering the imbalance, all the partitions are obtained with perfect or almost perfect size. In particular, both Zoltan RCB and Directed partitioning provide excellent results: each partition V_i has size equal to $|V_i|/\lceil|V|/k\rceil$, that is the desired threshold. ParMETIS blocks are slightly imbalanced, but the amount is negligible if compared to Zoltan RCB and Directed partitioning.

Lastly, consider Table 4.10, showing the average ratio of the number of elements in a partition border and the size of the partition. Directed partitioning ratios are placed between ParMETIS and Zoltan RCB values, meaning that the obtained partitions are patch-like and compact.

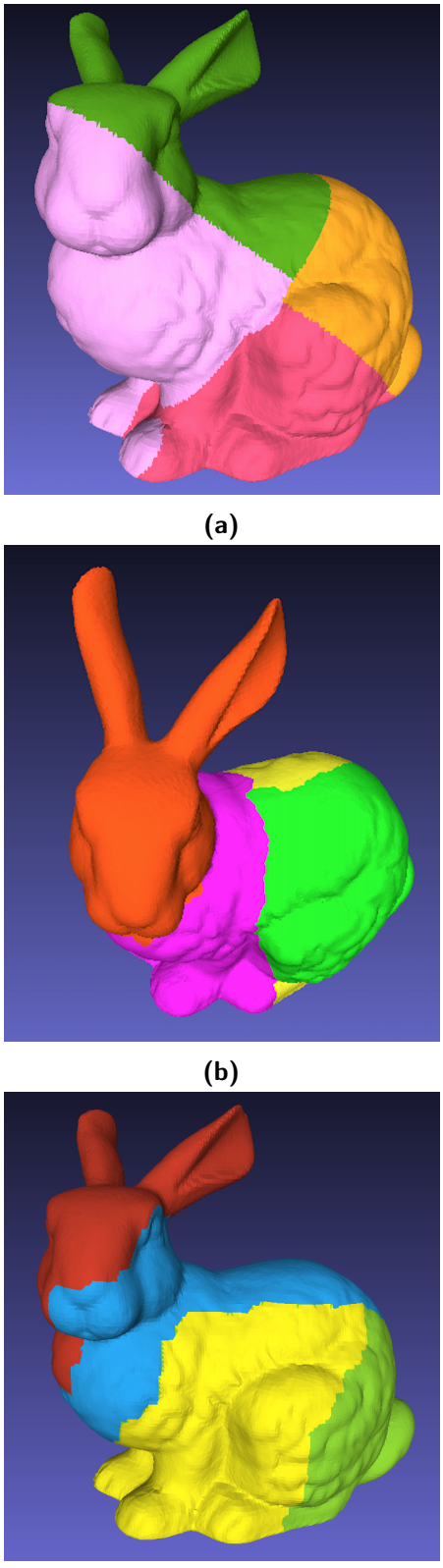
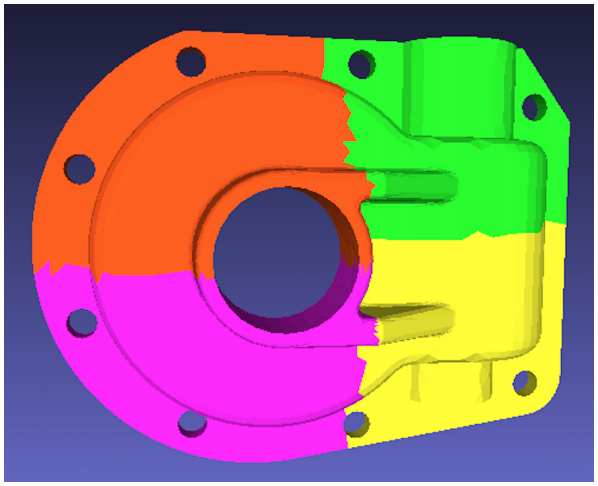
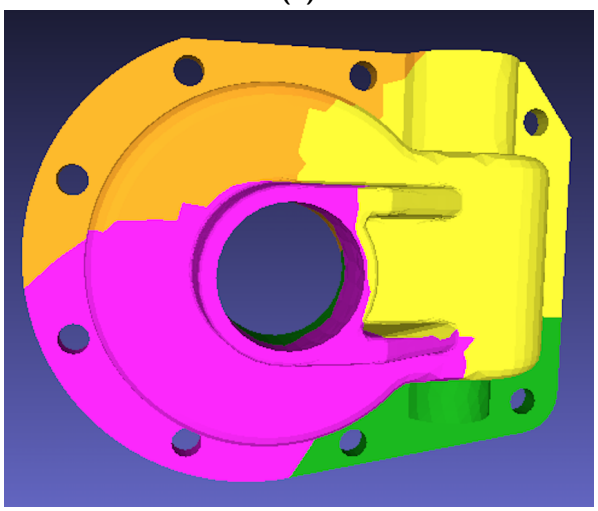


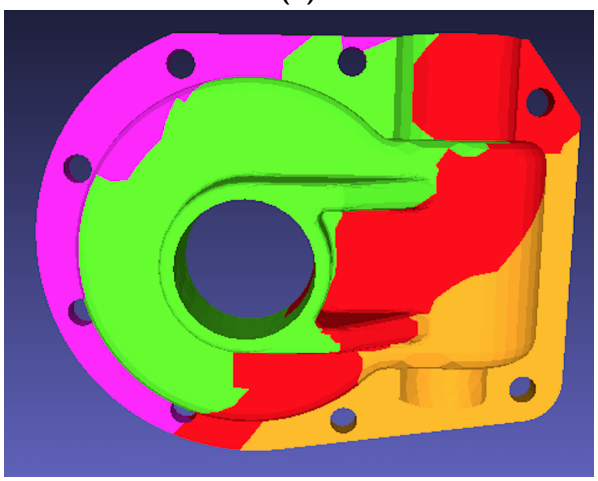
Figure 5.20: From top to bottom, left to right: Zoltan RCB, ParMETIS and Directed partitioning on the mesh bunny.



(a)

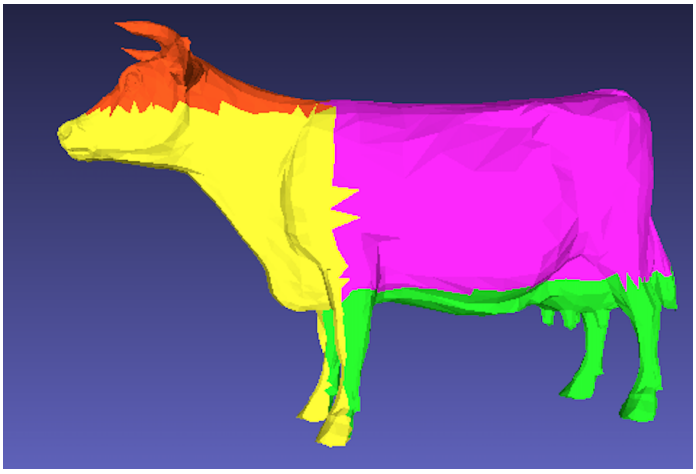


(b)

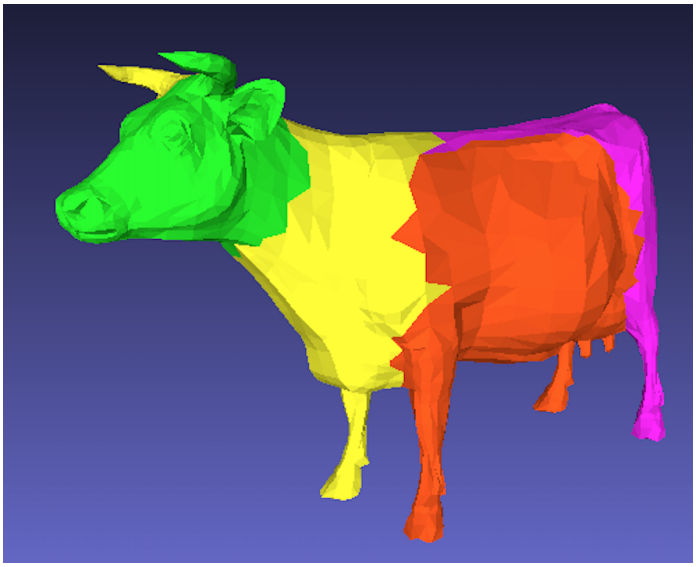


(c)

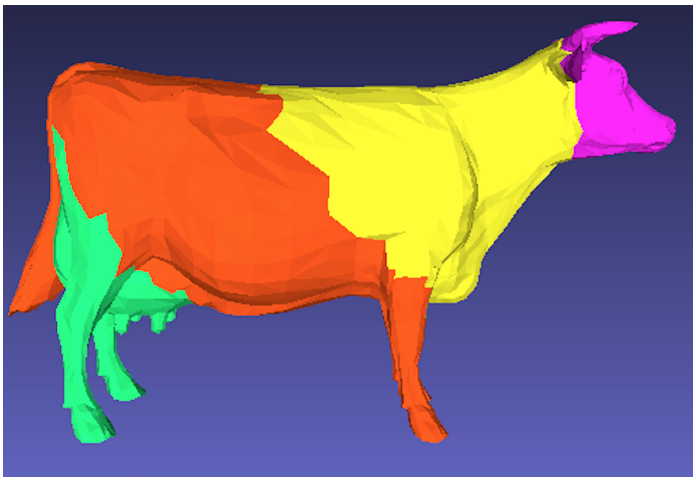
Figure 5.21: From top to bottom, left to right: Zoltan RCB, ParMETIS and Directed partitioning on the mesh casting.



(a)

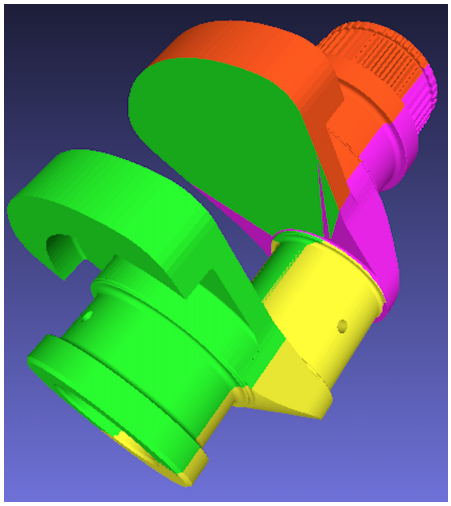


(b)

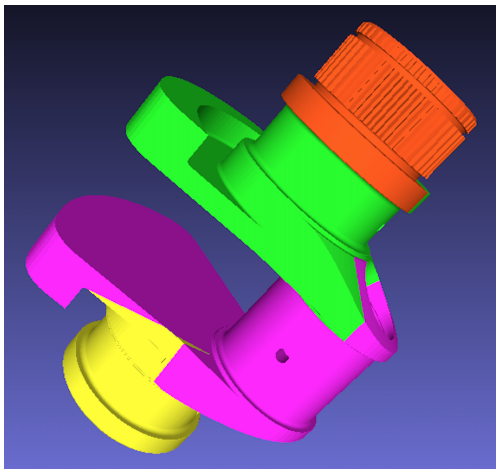


(c)

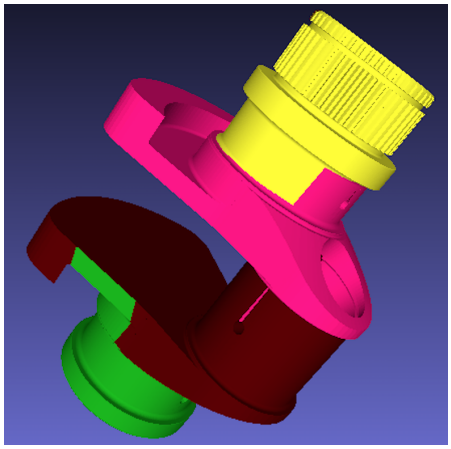
Figure 5.22: From top to bottom, left to right: Zoltan RCB, ParMETIS and Directed partitioning on the mesh cow.



(a)

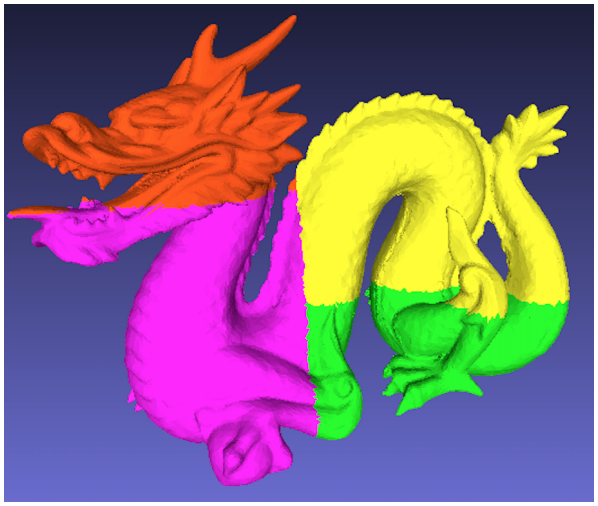


(b)

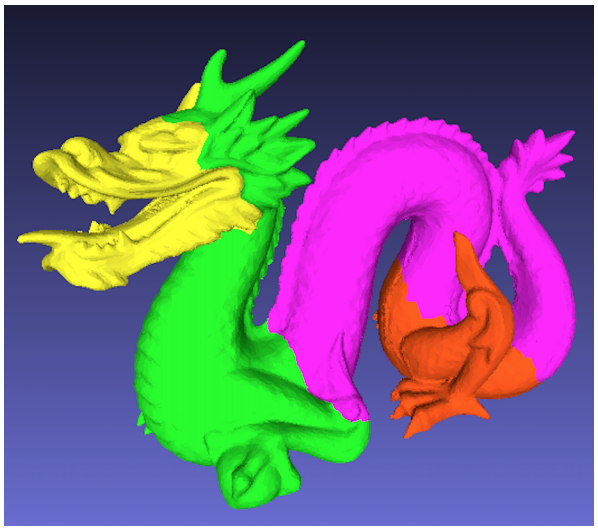


(c)

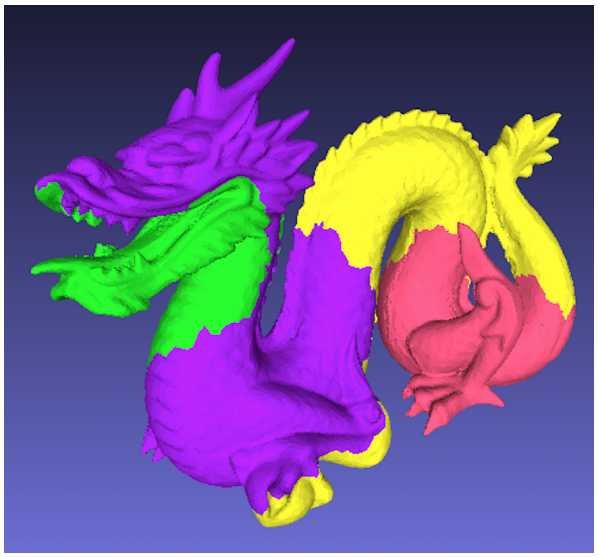
Figure 5.23: From top to bottom, left to right: Zoltan RCB, ParMETIS and Directed partitioning on the mesh crank.



(a)



(b)



(c)

Figure 5.24: From top to bottom, left to right: Zoltan RCB, ParMETIS and Directed partitioning on the mesh dragon.

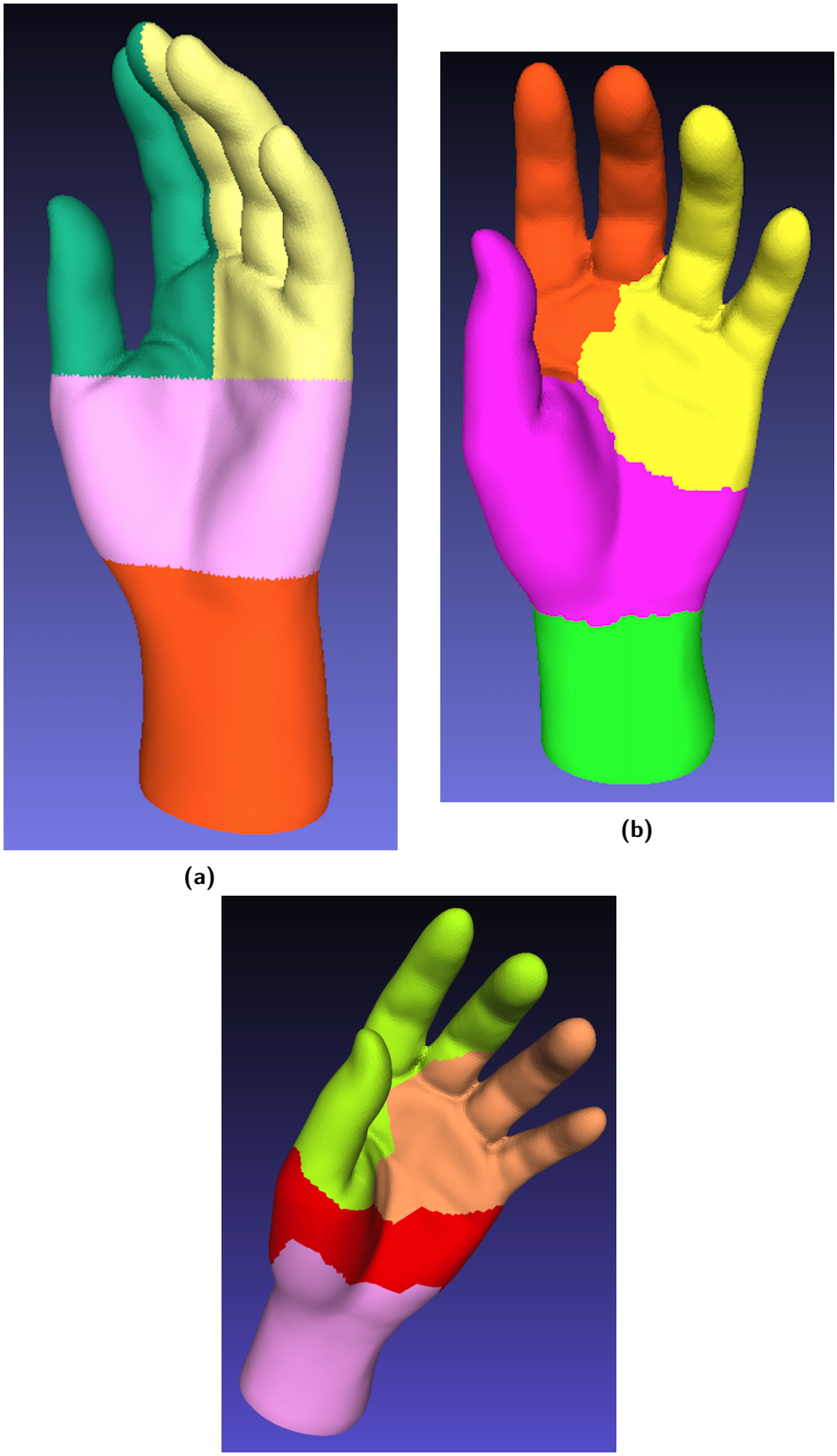
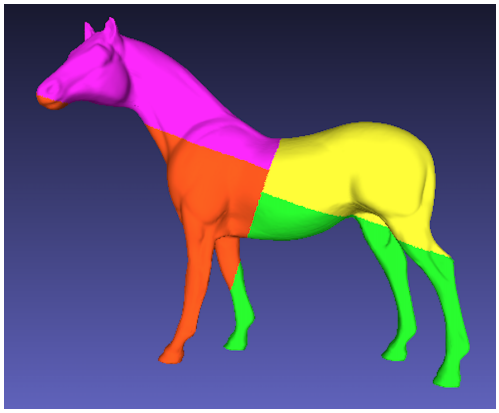
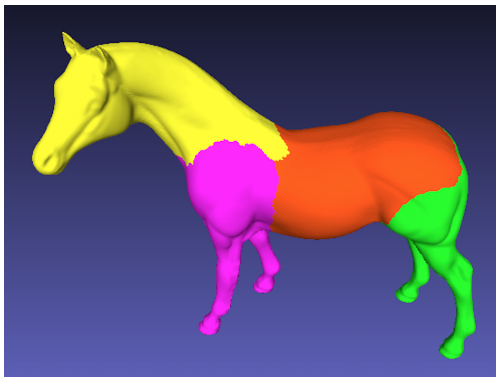


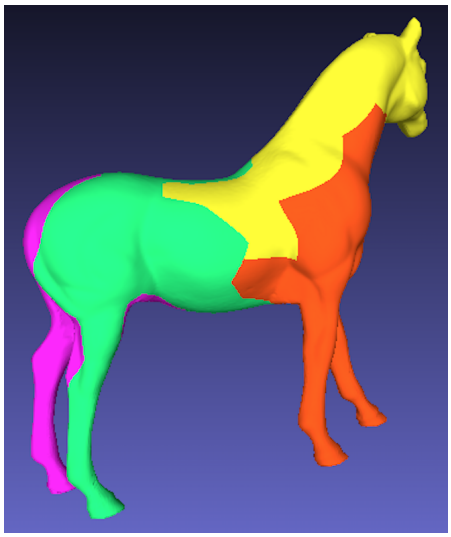
Figure 5.25: From top to bottom, left to right: Zoltan RCB, ParMETIS and Directed partitioning on the mesh hand.



(a)



(b)



(c)

Figure 5.26: From top to bottom, left to right: Zoltan RCB, ParMETIS and Directed partitioning on the mesh horse.

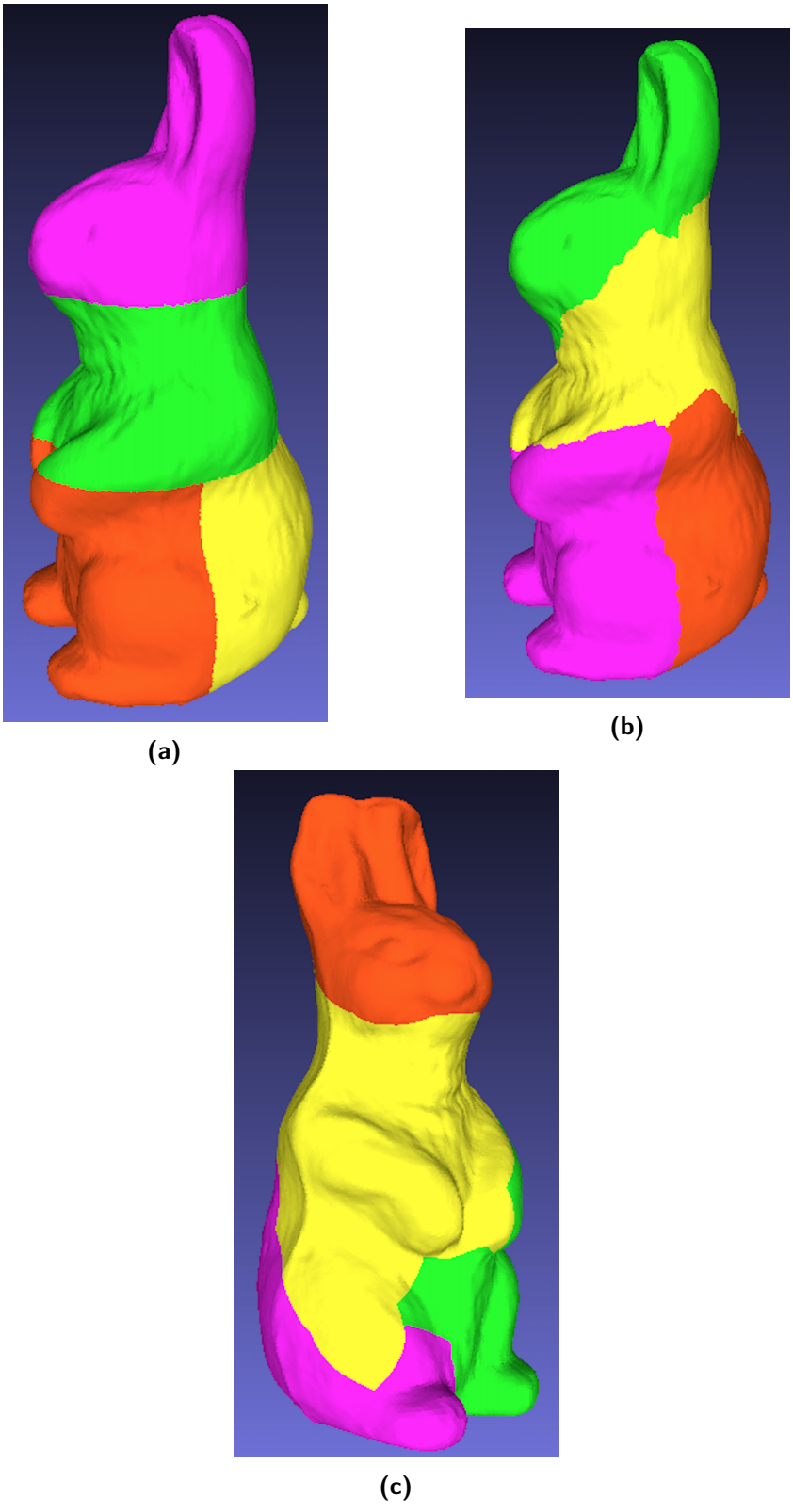


Figure 5.27: From top to bottom, left to right: Zoltan RCB, ParMETIS and Directed partitioning on the mesh rabbit.

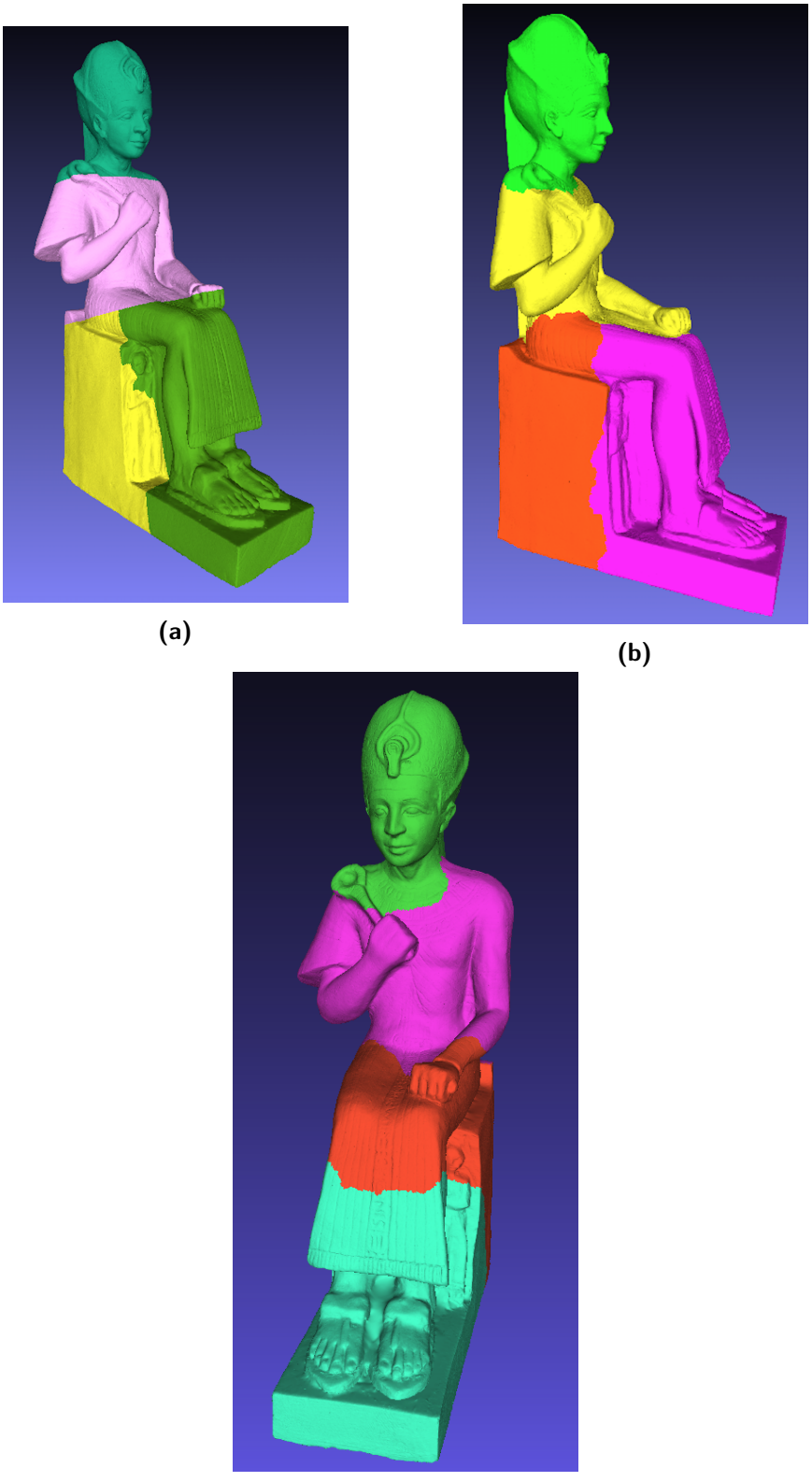
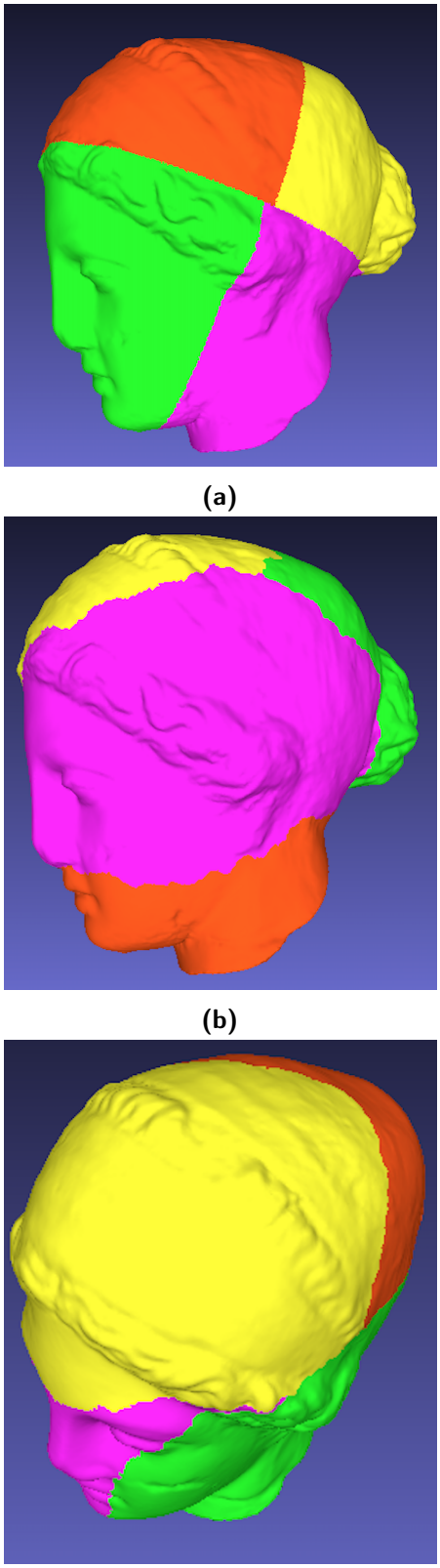


Figure 5.28: From top to bottom, left to right: Zoltan RCB, ParMETIS and Directed partitioning on the mesh ramesses.



(a) (b) (c)

Figure 5.29: From top to bottom, left to right: Zoltan RCB, ParMETIS and Directed partitioning on the mesh venus.

BUNNY	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	69666	2.564	1	1
ParMETIS		0.122	1	1
Directed part.		**0.0245 + 0.0175 + 0.0112 (0.0532)**	1	1

Table 5.1: Comparison between partitioning algorithms on the mesh bunny.

CASTING	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	10224	2.169	1	1
ParMETIS		0.0457	1	1
Directed part.		**0.00271 + 0.00216 + 0.00234 (0.00721)**	1	1

Table 5.2: Comparison between partitioning algorithms on the mesh casting.

COW	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	5804	2.267	1	1
ParMETIS		0.0441	1.028	0.972
Directed part.		**0.00231 + 0.00218 + 0.00176 (0.00625)**	1	1

Table 5.3: Comparison between partitioning algorithms on the mesh cow.

CRANK	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	100056	2.489	1	1
ParMETIS		0.202	1	1
Directed part.		**0.0368 + 0.0184 + 0.0156 (0.0708)**	1	1

Table 5.4: Comparison between partitioning algorithms on the mesh crank.

DRAGON	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	100000	2.189	1	1
ParMETIS		0.229	1.001	0.998
Directed part.		**0.0419 + 0.0324 + 0.0289 (0.103)**	1	1

Table 5.5: Comparison between partitioning algorithms on the mesh dragon.

HAND	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	72958	2.176	1	1
ParMETIS		0.187	1.001	0.997
Directed part.		**0.037 + 0.0118 + 0.0154 (0.0642)**	1	1

Table 5.6: Comparison between partitioning algorithms on the mesh hand.

HORSE	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	225280	2.443	1	1
ParMETIS		0.395	1.003	0.998
Directed part.		**0.0788 + 0.0511 + 0.0563 (0.186)**	1	1

Table 5.7: Comparison between partitioning algorithms on the mesh horse.

RABBIT	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	141312	2.491	1	1
ParMETIS		0.284	1.001	0.999
Directed part.		**0.0512 + 0.0313 + 0.0297 (0.112)**	1	1

Table 5.8: Comparison between partitioning algorithms on the mesh rabbit.

RAMESSES	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	1652528	7.929	1	1
ParMETIS		3.578	n.c.	n.c.
Directed part.		**0.758 + 0.319 + 0.359 (1.436)**	1	1

Table 5.9: Comparison between partitioning algorithms on the mesh ramesses.

VENUS	|Nodes|	Time (secs)	Max imb.	Min imb.
Zoltan RCB	201515	2.453	1	1
ParMETIS		0.423	1.003	0.998
Directed part.		**0.0754 + 0.0426 + 0.0404 (0.158)**	1	1

Table 5.10: Comparison between partitioning algorithms on the mesh venus.

	Zoltan RCB	ParMETIS	Directed part.
bunny	2.82	1.72	3.10
casting	8.23	5.27	7.39
cow	10.67	4.88	5.53
crank	2.92	0.94	3.01
dragon	2.46	0.80	2.95
hand	2.62	1.86	2.32
horse	1.57	0.87	1.08
rabbit	1.75	1.46	1.44
ramesses	0.44	0.29	0.38
venus	1.57	1.13	1.43

Table 5.11: Average percentages of border nodes with respect to the partitions size.

6

USE CASE AND AD-TREE PARTITIONING IMPROVEMENTS

In this chapter we present one of the possible applications of graph partitioning: 3D reconstruction. To obtain accurate representation of real world scenes, a reconstruction undergoes different steps of a pipeline: scene point estimate, 3D reconstruction (usually in the form of 3D surfaces, named meshes) and refinement of the created reconstruction. The last step is usually computationally expensive, since it is done on the whole mesh. Graph partitioning can be used to break it into multiple smaller parts and refine them separately and in parallel. Then, we discuss three possible improvements for the AD-tree partitioning algorithm presented in Chapter 4: a pre-processing tree rebalancing, an enhancement of the algorithm and a post-processing method to improve the balance of the obtained partitions.

6.1 USE CASE: 3D RECONSTRUCTION

3D Reconstruction represents a long-standing research topic in both Computer Vision and Robotics, used in different applications in the field of medicine, gaming, civil engineering, autonomous driving, tourism, etc. In the last decades, there was an important demand for 3D content for computer graphics, virtual reality and communication, further increasing the focus dedicated to the reconstruction problem.

In Computer Vision, researchers try to recover the poses of the camera used to obtain the scene images, together with the structure of the map, in a batch fashion, with Structure from Motion (SfM) algorithms. Many works in Robotics focus, instead, on the real-time creation of maps of large scale areas, having a camera navigating through the environment to be mapped, leading to the so called Visual Simultaneous Localization and Mapping (V-SLAM) algorithms. Simultaneous Localization and Mapping (SLAM) aims at rapidly understanding how the surrounding world looks like, chasing a trade-off between accuracy and computational feasibility.

SfM and Visual SLAM algorithms build a point-based map of the environment that is often very sparse, except in the case of recent semi-dense SLAM, such as LSD-SLAM [83]. Only few Dense-SLAM proposals, as DTAM [84], are able to recover a dense representation of the scene. However, they scale badly in large-scale environments.

Structure from Motion algorithms evolved into the so called Multi-View Stereo (MVS) algorithms that recover a dense accurate reconstruction of the

environment from a set of unordered images. MVS algorithms decouple the camera pose estimation and the model computation: they delegate the former task to an external algorithm, such as a SfM, and they focus on the latter to estimate a detailed model of the environment. As for the early SfM, Multi-View Stereo focuses on accuracy but it has limited its application only to offline batch settings.

Nowadays, the Multi-View Stereo community turned to volumetric and mesh-based representations, that was boosted thanks to the widespread availability of GPU hardware which enables massive parallel processing. Volumetric algorithms partition the scene into voxels or tetrahedra and estimate a subset representing the free space and a subset representing the matter. The boundary between free space and matter represents the final 3D model of the scene. Volumetric algorithms achieve very accurate results but their application is limited to small scenes, since their data structure is not scalable.

Very recent works [77–79, 81] represent the scene as a triangular manifold mesh, allowing to reconstruct large-scale scenes through continuous meshes. These are refined using photometric techniques, maximising the likelihood of the reconstruction according to the images. Their work can be divided in two contributions: first, they propose an incremental reconstruction from sparse points, obtaining a mesh; then, they explain how a mesh can be refined incrementally. The former step estimates a manifold mesh from the output of a Structure from Motion algorithm or any algorithm providing camera poses, sparse point clouds, and camera-to-point viewing rays (e.g., RGB-D and laser based systems). The latter step refines the resulting mesh according to the appearance provided by the images.

More in detail, their incremental pipeline combines three phases to reconstruct a photo-consistent model of the environment, represented in Figure 6.1: manifold reconstruction, mesh merging and windowed photo-metric refinement. The inputs of their proposed system are the 3D points position, the camera poses and the visibility rays, estimated by any incremental SfM or SLAM algorithm such as the method implemented in openMVG [85] or ORB-SLAM [86]. Every W frames the system outputs the available reconstruction.

The first step builds incrementally a manifold mesh from Structure from Motion 3D points and camera poses up to time t. In this step the manifold property is enforced, to consistently apply the mesh evolution process. The second step is dedicated to the windowed variational refinement of the current mesh, which evolves the surface to maximize the photo-consistency between pairs of cameras. The output of this module is the current incremental optimized mesh, i.e., the output of the whole system. In the third step the output of refinement and the new manifold are merged with a novel approach that keeps the manifold property and updates the topology. The

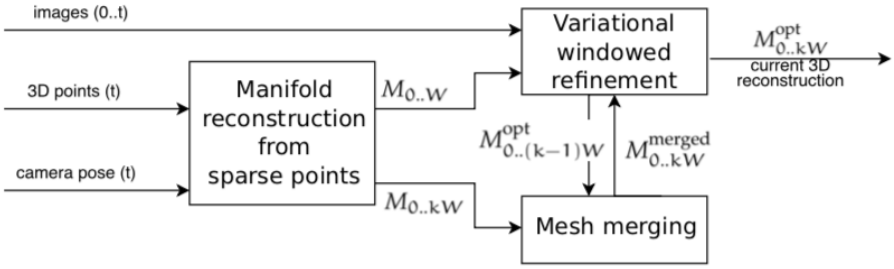


Figure 6.1: Pipeline of the 3D reconstruction method described in [77–79, 81]

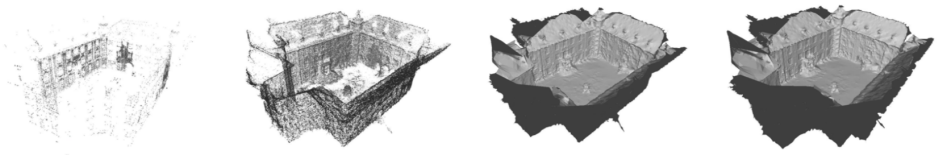


Figure 6.2: Complete 3D reconstruction pipeline. From left to right: sparse point estimate, densification of the points, manifold reconstruction and mesh refinement.

output of this module is the multi-resolution merged mesh that feeds the refinement module. After such multi-resolution merging step the refinement algorithm is able to jointly refine both the newly added region together with the refined part.

Instead of considering as input sparse points, one can also give a dense representation of the scene using sweeping methods, as in [87]. Although very costly, this improvement allows to obtain a better 3D reconstruction. Considering all the improvements, the general reconstruction pipeline is represented in figure 6.2 (mesh derived from the castle-P30 sequence of the EPFL dataset and built with the mentioned algorithms). After having estimated a sparse point cloud from a SfM or SLAM algorithm, the points are densified using sweeping methods, obtaining point clouds with a large amount of elements. From these, an accurate manifold reconstruction is obtained, successively refined through photometric or semantic techniques.

Although the output reconstruction is very detailed, the whole procedure is extremely slow, especially the densification and refinement steps. The time required to refine a mesh depends on its size, since it involves the movement of vertices to maximise the likelihood of reconstruction with respect to the images.

Instead of considering the whole mesh for the refinement phase, the pipeline can be improved by breaking the initial reconstruction and feed each part to multiple refinement modules, in parallel. Using either AD-tree partitioning or Directed partitioning, the mesh is divided in k parts. All the sub-meshes are then refined, in parallel and independently from each other. Lastly, the

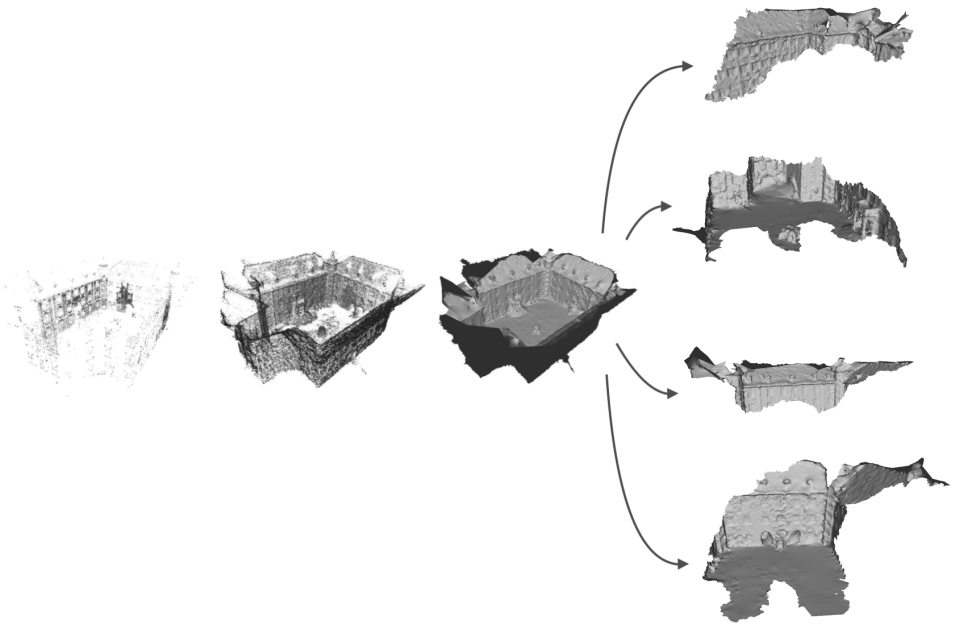


Figure 6.3: New proposed 3D reconstruction pipeline. From left to right: sparse point estimate, densification of the points, manifold reconstruction and mesh refinement on the partitions.

refined sub-meshes are stitched together to form again the entire mesh. An example is given in Figure 6.3, where the considered mesh is divided in four parts and each one is refined. Each step of the refinement on the full reconstruction takes around **180 seconds**. The mesh partition takes at most **1 or 2 seconds**, and the refinement on each derived sub-mesh is done in around **100 seconds** for each iteration, far less than the time required for the whole mesh.

6.2 IMPROVEMENTS OF AD-TREE PARTITIONING

In this section, we propose three enhancements for the AD-tree partitioning algorithm, to be implemented in future works. They all rely on the descendants graph component of the main data structure, using different working principles to improve the partitions balance.

The first enhancement consists in modifying the AD-tree before the partitioning, and is named *adoption*. The second improvement, *greedy T-sum*, works instead during the partitioning algorithm, to support the SearchDescendants routine of Algorithm 11, described in Section 4.4, Chapter 4. The third and last improvement, called *exchange* is a post-processing refinement:

once the partitions are obtained, they request and concede chunks of trees to increase the overall balance of the results.

6.2.1 *Adoption*

For a node in the AD-tree, children, descendants and relatives represent the adjacency in the original graph. The way these three parts interact is determined by the order imposed by the construction algorithm. One can, however, perform an operation that allows to change the roles: adoption.

A node with descendants can adopt one or more of them changing their ownership. Suppose to have a node v, whose descendant is u. If v adopts u, the following actions must be done:

1. u is removed from the descendants of v
2. u is added to the children of v
3. u's level and the levels of all the nodes departing from it are upgraded according to the level of v
4. Calling up the parent of u, up removes u from its children
5. If the new level of u is greater than the level of up, the latter becomes ancestor of the first (and u is inserted in up descendants)
6. If the new level of u is lesser than the level of up, the latter becomes descendant of the first (and u is inserted in up ancestors)
7. If the new level of u differs from its old level, all the nodes departing from it must change their descendants and ancestors accordingly to their new level (maintaining or inverting it)
8. Ariadne's tree is rebuilt for the whole AD-tree
9. If the new level of u is equal to the level of up, their relationship is determined by their visiting order, dictated by the new Ariadne's tree, with the usual convention that the first node to be visited is the ancestor

Conversely, a node with ancestors can be adopted by one of them, following the same operations. The Ariadne's tree reconstruction should be done just one time, after having moved multiple nodes, because it has not any purpose during the adoption phase: the only elements considered are ancestors, descendants and children (so the standard tree and the descendants graph).

Let's consider, again, the graph and corresponding LtR AD-tree in Figure 6.4. Suppose that node 7 wants to adopt node 6. The node is removed from the children of node 5 and inserted into the children of node 7. 6 becomes

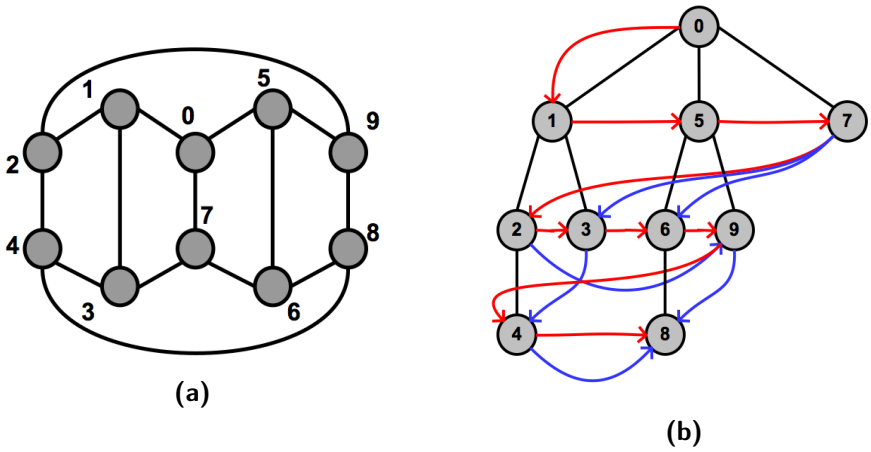


Figure 6.4: Graph and corresponding AD-tree used in the adoption example.

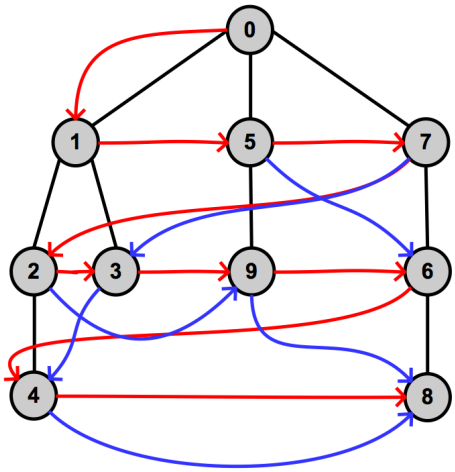


Figure 6.5: AD-tree obtained after the adoption of node 6 by node 7.

descendant of its old parent, 5, and since its level does not change, no update must be done on the nodes departing from it. Lastly, Ariadne's tree is reconstructed, obtaining the new AD-tree represented in Figure 6.5.

One can decide if an adopted node can be adopted again by a node belonging to a different branch, or if it can itself adopt. This means that adoption is bidirectional: it can be done by nodes belonging to all branches (moving horizontally in the tree) or by nodes belonging to adopted subtrees (moving vertically).

Indeed, adoption comes in different flavours, but should not be abused or done indiscriminately. A good practice is filling the ranks: a node should be adopted only if it is an only child or has only one sibling; a node should adopt only if it has one or two descendants. First, this approach reduces the computational time derived from adoption, since it greatly reduces the

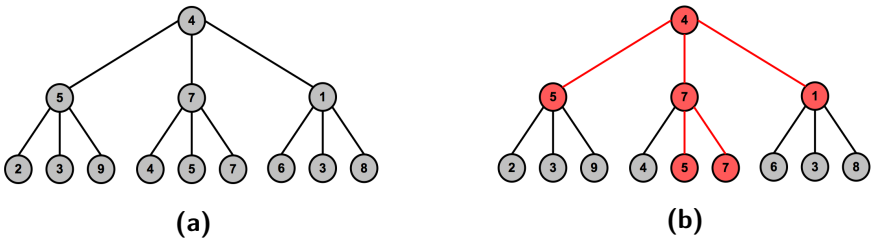


Figure 6.6: Tree with values and one of the possible subtrees with sum 29.

number of candidates for the procedure. Second, it balances the tree, moving thin branches and creating nodes with high fanout, leading to more balanced partitions.

6.2.2 *Greedy T-sum algorithm*

In Section 4.4 of Chapter 4, the working principle of the AD-tree partitioning algorithm has been described. One particular subroutine, SearchDescendants, is used to find a path that exceeds the cut threshold. Although this procedure is fast, it is limited in the search space: suppose to have a very dense descendants graph, searching for only a path would waste many possibilities to find better cuts.

Instead of considering a path, one can search a subtree with same root, such that the sum of the values of all its nodes is the minimum sum among all the possible subtrees, greater than a fixed threshold. If the selected subtree has value $\mathtt{val}$ that is greater than the threshold, than there is no other subtree whose value stands between the threshold and $\mathtt{val}$. For example, consider the tree in Figure 6.6a. If the threshold is 29, one of the many subtrees to have this value is represented in Figure 6.6b. The exact solution can be found enumerating all the possible subtrees and stopping as soon as a subtree with the desired value is found. This solution is acceptable for short trees, but it becomes too difficult to compute for trees with many levels (as in the AD-tree).

Without loss of generality, let's consider a complete tree, where each node has the same degree D. If the tree consists only in the root, only one possible subgraph can be extracted: the root itself. If the three has two levels, the number of possible subgraphs, named N_2, is the number of combinations of the nodes in the tree, computed as

$$N_2 = \sum_{k=0}^{D} \binom{D}{k} = 2^D$$

It is well known [73] that in case of infinite trees of degree D, the number of rooted subtrees, with given size u, is given by

$$\frac{1}{(D-1)u+1}\binom{Du}{u}$$

For finite trees, an expression of the number of rooted subtrees is provided, without proving it but giving an intuitive explanation. Let v be some node which is not a leaf, and let its children be $v_1, v_2, \ldots, v_d$. Suppose that the number of subtrees of the maximal subtree rooted at each child (i.e., which include these but no parent) has already been counted. Considering that these numbers are always the same, n, since each node has the same number of elements, then the maximum number of subtrees rooted at v is $(1+n)^D$. This concept is generalisable to the total number of subtrees rooted in a tree, and can be explained recursively.

- The tree is only formed by the root: the only possible rooted subtree is the tree itself, hence $N_1 = 1$.
- The tree has two levels, meaning that the number of roots is given by $(1+n)^D$, where n is the number of subtrees rooted in the leaves. n coincides with N_1, because it represents the first case (trees with only one node), hence $N_2 = (1+1)^D = 2^D$, as seen before.
- The tree has L levels, meaning that the number of roots is given by $(1+N_{L-1})^D$, where N_{L-1} is the number of subtrees rooted in the children of the root (that corresponds to subtrees of $L-1$ levels).

Summing up, given a complete tree with level L and each node having D children, the number of rooted subtrees is given by:

$$\begin{cases} N_1 = 1 \\ N_L = (1+N_{L-1})^D \end{cases}$$

This number is exponential with respect to the levels of the tree, thus making unfeasible the enumeration of subtrees.

We propose a greedy algorithm to find in feasible time an acceptable solution (minimum sum greater than a threshold T), called greedy T-sum. Given a tree of L levels, the algorithm proceeds in the following way.

- A window W of size L_W, that focuses on the portion of the tree enclosed by levels 0 and $L_W - 1$, is defined. L_W depends on D, but usually small values like 2 or 3 are used.
- On the portion of the tree enclosed by this window, the algorithm enumerates all the possible rooted subtrees, ranking them by sum of the nodes' values and selecting the best three.

- All the subtrees selected that do not have the same depth of the window are directly considered candidate solutions.
- The window is shifted by $L_W - 1$ levels, such that the leaves of the selected subtrees are now roots of the subtrees defined by the moved window.
- For each root, enumerate all the possible subtrees and as before, rank them, selecting the best three among all the subtrees.
- The last three steps are repeated until either the window reaches the end of the tree or the exact solution is found.

The algorithm is not guaranteed to provide the best solution, but it greatly reduces the number of enumerations.

Let's end the subsection with an example of the algorithm application, using the tree represented before (in Figure 6.6a) and considering the desired T-sum of 29 and a window of size 2 (Figure 6.7a). The two best solutions are considered, with value respectively of 16 and 17 (Figures 6.7b and 6.7c). The window is then shifted, such that the leaves of the subtrees considered are now roots of new subtrees (Figure 6.7d). All the subtrees are enumerated and the best two solutions, that also are the exact T-sum, are found (Figures 6.7e and 6.7f).

6.2.3 *Exchange*

A more subtle way to exploit the adoption technique described before is to consider partitions as big clusters, connected by edges crossing the cut border. When a tree is partitioned into multiple groups, each block has edges (children, descendants and relatives) going both inside and outside the partition itself (see Figure 6.8). While edges that belong to it are not so useful, the ones crossing the cuts are the most interesting, because they allow an interaction between partitions.

Since the groups have different sizes, one can try to balance them having the partitions make requests and donations, following the idea behind the refinement phase in multi-level graph partitioning approaches. If a partition size is less than the desired value, it searches for possible adoptions in the partitions connected to it by descendants edges. Vice versa, if a partition size is greater than the desired value, it offers subtrees rooted in nodes that have ancestors belonging to different partitions (example in Figure 6.9).

While this method sounds more appealing than the chaotic adoption described before, it requires a global planning due to the fact that we need to maximise the balance of all the partitions (not just of one). This involves considering all the crossing edges: for large graphs it is highly probable that the

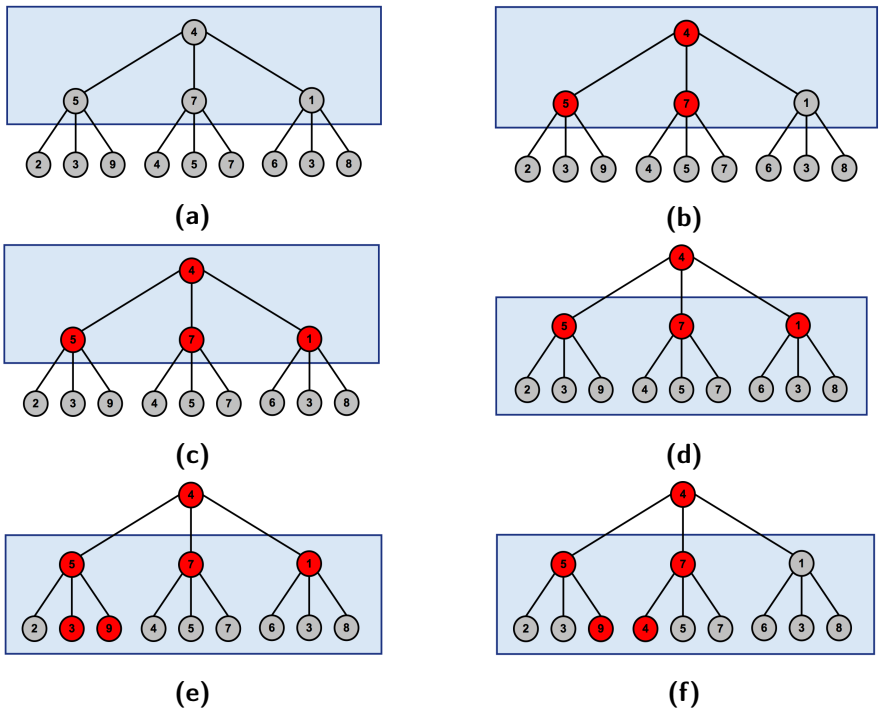


Figure 6.7: Example of how the greedy T-sum algorithm works.

number of such edges is elevate, leading to an increased execution time. The exchange happens only with descendants and ancestors and not with children. Indeed, while the edges associated to the last two can link any point inside a partitions, children edges always connect a leaf of one partition to a root of another group, because of how the partitioning algorithm works. For this reason they are useless in the exchange procedure: they cannot be exchanged because it would mean destroying an entire block.

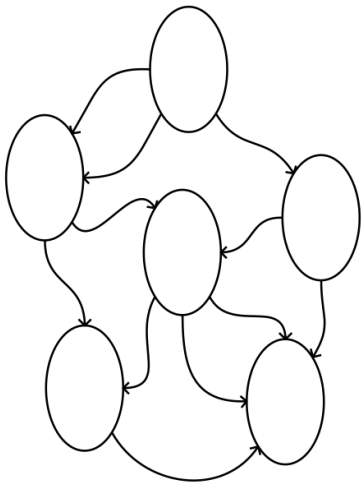


Figure 6.8: High level view of a partitioned AD-tree: partitions are connected by edges representing descendants, ancestors and children.

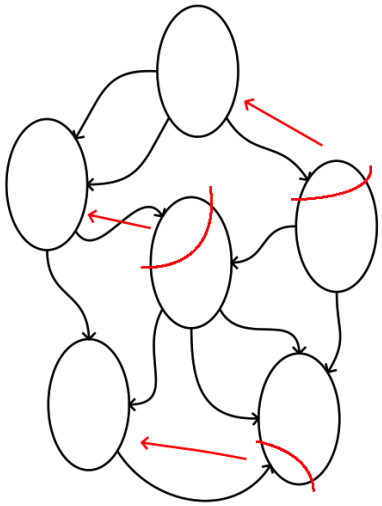


Figure 6.9: Example of how sub-trees are exchanged among partitions to achieve more balance.

7

CONCLUSIONS AND FUTURE WORKS

In this thesis, we proposed three new graph partitioning algorithms. Although each algorithm differs from the others by performance, usage, output quality and working principle, they all allow to perform efficient and fast partitioning on graphs.

The first algorithm is designed for hybrid clustering: it exploits the similarities between nodes to divide a graph into k parts. It has the purpose of breaking a graph into k blocks, typical requirement of graph partitioning algorithms, such that the nodes within them are highly similar, as any clustering method. For this reason, the algorithm, Label clustering, can be considered a hybrid between partitioning and clustering approaches.

The algorithm structure resembles the one adopted by multi-level partitioning algorithms, that represent the current stream of scientific works. A graph is first reduced using two procedures, the first applied directly and the second repeated more than once. The first consists into clustering together all the nodes having a very high similarity. The second is a coarsening algorithm, applied only if the graph derived from the first procedure has a high amount of nodes. On the small coarsened graph, a direct partitioning method is used to obtain k blocks, that are than backtracked into the original graph, in order to assign each node to one of the k partitions.

Each phase of the Label clustering algorithm has its advantages. The first, mandatory, reduces a labelled graph to a new graph where each node is adjacent to elements with different label (or adjacent to nodes having similarity below a fixed threshold). This procedure is very fast, reducing graphs with millions of nodes and edges in the order of seconds. The second, graph coarsening, further decreases the size of the reduced graph, if it exceeds a value. Exploiting the neighbourhood similarities, it clusters nodes locally, retaining organized groups. The last phase, Normalised Spectral Clustering is applied on the smallest coarsened graph to obtain a high quality partitioning.

The second proposed algorithm, that is a form of tree partitioning, relies on an augmented data structure to represent graphs using trees. This data structure, the AD-tree, consists in three components:

- the standard tree representation of a graph, computed using the breadth-first visiting algorithm on graphs
- the descendants graph, containing directed edges of the graph that are not considered during the arborescence creation

- Ariadne's tree, that is the sequence of nodes connected through directed edges during the breadth first visiting

The AD-tree partitioning algorithm works on the new defined data structure, starting from the last inserted element and backtracking through Ariadne's tree, propagating the values (number of elements in the subtree rooted in a node) of the nodes to their parents. When the value of a node exceeds the cut threshold, all the nodes belonging to the rooted subtree in it, considering both the normal tree and the descendants graph, are assigned to a partition. This is done until k cuts are made, obtaining as many blocks.

The algorithm shows an incredible speed, surpassing all the compared algorithms, although being a serial method. Not just the partitioning is fast, but also the tree creation, dividing graphs in one order of magnitude less time than the state of the art methods. Moreover, on average the resulting partitions have a small amount of border elements, meaning that the shapes tend to be compact and patch-like, desirable property when partitioning graphs derived by geometric entities, such as meshes. Lastly, the partitions obtained from AD-tree partitioning are compact and do not have more than one connected component.

The third and last proposed algorithm, Directed partitioning, is used mainly for graphs derived by regular structures, as meshes or networks representing particular situations (for example, if one wants to describe the interactions and purpose of the cells of a beehive). The idea behind it is straightforward: given a graph, explore its nodes with known algorithms going counterclockwise,until the desired count is reached. This is done to generate regular structures that translates into patch-like visual structures on the graph (groups of connected nodes). The exploration procedure creates holes, that are detected and filled, assigning them to the set of explored nodes. Then, to maintain the perfectly balanced property, granted by the visiting algorithm, some nodes of the partition are unassigned to it.

For small k, the algorithm is faster than the state of the art methods. Moreover, independently from k, it produces perfectly balanced partitions whose number of border elements is a small fraction of their size. This means that not only Directed partitioning divides a graph in blocks of equal size, but these partitions are obtained with a very low number of cuts in the original graph.

Each of the three presented algorithm can be improved in different way. Considering Label clustering, the coarsening phase can be parallelised, to further increase the algorithm's speed, and a different heuristic can be used to group together the nodes. Aggregation by maximum similarity, the method adopted to cluster pairs of nodes, is easy to implement, but is limited to a node surroundings, not considering the whole connectivity of the graph. Moreover, a refinement step should be inserted after the direct partitioning, to move group of nodes increasing the internal weight of all the clusters.

Lastly, also the first reduction of the graph can be parallelised, since it is based on depth-first search, in order to handle large graphs and increase the scalability of Label clustering.

While the AD-tree partitioning is very efficient and fast, it divides a graph in slightly imbalanced partitions. Moreover, the algorithm works on an auxiliary data structure, so to the partitioning time one need to sum also the time spent in creating the AD-tree. Three improvements can be done on the algorithm. First, the structure creation should be enhanced increasing the degree of parallelisation, already present in a raw form. This is needed to reduce the computational impact on the whole algorithm. Then, the partitioning procedure can be revisited to increase the quality of the partitions, for example introducing new checks on the nodes or modifying already existent routines. Lastly, a refinement done after the partitioning could greatly improve the blocks quality, both in terms of maximum imbalance and minimum cut (or maximum intra-partition weights).

Particularly interesting is another usage of the algorithm. In the thesis, the concept of partitions was usually associated to the number of nodes. Tree partitioning hides a further application. The value that is propagated is not necessarily the number of nodes in the considered rooted subtree, but can have many other interpretations, depending on the context of usage. If one considers surfaces, the value can be an area or a dihedral angle; if transportation and allocation problems are represented, the value can mean something like the available space or quantity of a site, and so on.

Lastly, Directed partitioning results heavily depend on the structure of the considered graph and the initial node selection to start the partitioning. The algorithm can be improved by modifying the procedures used in the various steps. For example hole detection is quite expensive, since it involves exploring the half of the graph that does not belong to the partition. Moreover, instead of bisection, the algorithm can be adapted to partition directly the graph into k perfectly balanced blocks.

We conclude the thesis focusing on the use case presented in Chapter 6. Refining the partitions instead of the complete mesh proved to be less computationally expensive and more accurate (in terms of new vertices created on the mesh) than refining the full mesh. It is still required to implement an efficient form of stitching, that should preserve the manifold property of the initial mesh.

REFERENCES

[1] Johannes Lutz Schönberger and Jan-Michael Frahm. "Structure-from-Motion Revisited." In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016 (cit. on pp. v, 69).

[2] Johannes Lutz Schönberger, Enliang Zheng, Marc Pollefeys, and Jan-Michael Frahm. "Pixelwise View Selection for Unstructured Multi-View Stereo." In: *European Conference on Computer Vision (ECCV)*. 2016 (cit. on pp. v, 69).

[3] Christoph Strecha, Wolfgang Von Hansen, Luc Van Gool, Pascal Fua, and Ulrich Thoennessen. "On benchmarking camera calibration and multi-view stereo for high resolution imagery." In: *2008 IEEE Conference on Computer Vision and Pattern Recognition*. Ieee. 2008, pp. 1–8 (cit. on pp. v, 69, 116).

[4] Kai Wang, Guillaume Lavoué, Florence Denis, Atilla Baskurt, and Xiyan He. "A benchmark for 3D mesh watermarking." In: *2010 Shape Modeling International Conference*. IEEE. 2010, pp. 231–235 (cit. on pp. v, 104, 137).

[5] Andreas Geiger, Philip Lenz, and Raquel Urtasun. "Are we ready for autonomous driving? the kitti vision benchmark suite." In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2012, pp. 3354–3361 (cit. on p. v).

[6] Robert Tarjan. "Depth-first search and linear graph algorithms." In: *SIAM journal on computing* 1.2 (1972), pp. 146–160 (cit. on p. 14).

[7] Rod Hynes. "A new class of set union algorithms." PhD thesis. National Library of Canada= Bibliothèque nationale du Canada, 1999 (cit. on p. 14).

[8] Francisco Pedroche, Miguel Rebollo, Carlos Carrascosa, and Alberto Palomares. "L-RCM: a method to detect connected components in undirected graphs by using the Laplacian matrix and the RCM algorithm." In: *arXiv preprint arXiv:1206.5726* (2012) (cit. on p. 14).

[9] Ali Varamesh and Mohammad Kazem Akbari. "Fast detection of connected components in large scale graphs using mapreduce." In: *IOSR Journal of Engineering* 4 (2014), pp. 35–42 (cit. on p. 14).

[10] Alessandro Lulli, Emanuele Carlini, Patrizio Dazzi, Claudio Lucchese, and Laura Ricci. "Fast connected components computation in large graphs by vertex pruning." In: *IEEE Transactions on Parallel and Distributed systems* 28.3 (2017), pp. 760–773 (cit. on p. 14).

[11] Luciano da Fontoura Costa, Osvaldo N Oliveira Jr, Gonzalo Travieso, Francisco Aparecido Rodrigues, Paulino Ribeiro Villas Boas, Lucas Antiqueira, Matheus Palhares Viana, and Luis Enrique Correa Rocha. "Analyzing and modeling real-world phenomena with complex networks: a survey of applications." In: *Advances in Physics* 60.3 (2011), pp. 329–412 (cit. on p. 15).

[12] Michael R Garey, David S Johnson, and Larry Stockmeyer. "Some simplified NP-complete problems." In: *Proceedings of the sixth annual ACM symposium on Theory of computing*. ACM. 1974, pp. 47–63 (cit. on p. 17).

[13] Konstantin Andreev and Harald Racke. "Balanced graph partitioning." In: *Theory of Computing Systems* 39.6 (2006), pp. 929–939 (cit. on p. 17).

[14] Horst D Simon. "Partitioning of unstructured problems for parallel processing." In: *Computing systems in engineering* 2.2-3 (1991), pp. 135–148 (cit. on pp. 19, 30, 35).

[15] Roy D Williams. "Performance of dynamic load balancing algorithms for unstructured mesh calculations." In: *Concurrency: Practice and experience* 3.5 (1991), pp. 457–481 (cit. on pp. 19, 30).

[16] Charbel Farhat and Michel Lesoinne. "Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics." In: *International Journal for Numerical Methods in Engineering* 36.5 (1993), pp. 745–764 (cit. on p. 19).

[17] Gary L Miller, S-H Teng, and Stephen A Vavasis. "A unified geometric approach to graph separators." In: *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*. IEEE. 1991, pp. 538–547 (cit. on p. 20).

[18] John R Gilbert, Gary L Miller, and Shang-Hua Teng. "Geometric mesh partitioning: Implementation and experiments." In: *SIAM Journal on Scientific Computing* 19.6 (1998), pp. 2091–2110 (cit. on p. 20).

[19] Hans Sagan. "Hilbert's space-filling curve." In: *Space-filling curves*. Springer, 1994, pp. 9–30 (cit. on p. 21).

[20] Gerhard Zumbusch. *Parallel Multilevel Methods: Adaptive Mesh Refinement and Loadbalancing*. Springer Science & Business Media, 2012 (cit. on p. 21).

[21] Stefan Schamberger and Jens-Michael Wierum. "A Locality Preserving Graph Ordering Approach for Implicit Partitioning: Graph-Filing Curves." In: *ISCA PDCS*. 2004, pp. 51–57 (cit. on p. 21).

[22] Moritz von Looz, Charilaos Tzovas, and Henning Meyerhenke. "Balanced k-means for Parallel Geometric Partitioning." In: *arXiv preprint arXiv:1805.01208* (2018) (cit. on p. 23).

[23] Mehmet Deveci, Sivasankaran Rajamanickam, Karen D Devine, and Ümit V Çatalyürek. "Multi-jagged: A scalable parallel spatial partitioning algorithm." In: *IEEE Transactions on Parallel and Distributed Systems* 27.3 (2016), pp. 803–817 (cit. on p. 23).

[24] Shad Kirmani and Padma Raghavan. "Scalable parallel graph partitioning." In: *Proceedings of the international conference on high performance computing, networking, storage and analysis*. ACM. 2013, p. 51 (cit. on p. 23).

[25] Lars Hagen and Andrew B Kahng. "New spectral methods for ratio cut partitioning and clustering." In: *IEEE transactions on computer-aided design of integrated circuits and systems* 11.9 (1992), pp. 1074–1085 (cit. on p. 25).

[26] Jianbo Shi and Jitendra Malik. "Normalized cuts and image segmentation." In: *Departmental Papers (CIS)* (2000), p. 107 (cit. on p. 25).

[27] Ulrike Von Luxburg. "A tutorial on spectral clustering." In: *Statistics and computing* 17.4 (2007), pp. 395–416 (cit. on p. 25).

[28] Boaz Nadler and Meirav Galun. "Fundamental limitations of spectral clustering." In: *Advances in neural information processing systems*. 2007, pp. 1017–1024 (cit. on p. 26).

[29] Lihi Zelnik-Manor and Pietro Perona. "Self-tuning spectral clustering." In: *Advances in neural information processing systems*. 2005, pp. 1601–1608 (cit. on p. 26).

[30] Wen-Yen Chen, Yangqiu Song, Hongjie Bai, Chih-Jen Lin, and Edward Y Chang. "Parallel spectral clustering in distributed systems." In: *IEEE transactions on pattern analysis and machine intelligence* 33.3 (2011), pp. 568–586 (cit. on p. 27).

[31] Yeqing Li, Feiping Nie, Heng Huang, and Junzhou Huang. "Large-scale multi-view spectral clustering via bipartite graph." In: *Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015 (cit. on p. 27).

[32] Nicolas Tremblay, Gilles Puy, Rémi Gribonval, and Pierre Vandergheynst. "Compressive spectral clustering." In: *International Conference on Machine Learning*. 2016, pp. 1002–1011 (cit. on p. 27).

[33] Uri Shaham, Kelly Stanton, Henry Li, Boaz Nadler, Ronen Basri, and Yuval Kluger. "Spectralnet: Spectral clustering using deep neural networks." In: *arXiv preprint arXiv:1801.01587* (2018) (cit. on p. 27).

[34] Stephen T Barnard and Horst D Simon. "Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems." In: *Concurrency: Practice and experience* 6.2 (1994), pp. 101–117 (cit. on p. 27).

[35] Bruce Hendrickson and Robert Leland. "A multi-level algorithm for partitioning graphs." In: (1995) (cit. on pp. 27, 33).

[36] Una Benlic and Jin-Kao Hao. "A multilevel memetic approach for improving graph k-partitions." In: *IEEE Transactions on Evolutionary Computation* 15.5 (2011), pp. 624–642 (cit. on pp. 27, 28).

[37] Alan J Soper, Chris Walshaw, and Mark Cross. "A combined evolutionary search and multilevel optimisation approach to graph-partitioning." In: *Journal of Global Optimization* 29.2 (2004), pp. 225–241 (cit. on p. 27).

[38] George Karypis and Vipin Kumar. "METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0." In: (1995) (cit. on pp. 28, 37, 104).

[39] Peter Sanders and Christian Schulz. "Engineering multilevel graph partitioning algorithms." In: *European Symposium on Algorithms*. Springer. 2011, pp. 469–480 (cit. on pp. 29, 34, 35, 38, 104).

[40] Peter Sanders and Christian Schulz. "KaHIP v2. 0–Karlsruhe High Quality Partitioning–User Guide." In: *arXiv preprint arXiv:1311.1714* (2013) (cit. on pp. 29, 35, 38).

[41] Jens Maue and Peter Sanders. "Engineering algorithms for approximate weighted matching." In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2007, pp. 242–255 (cit. on pp. 28, 29).

[42] Ilya Safro, Peter Sanders, and Christian Schulz. "Advanced Coarsening Schemes for Graph Partitioning." In: *International Symposium on Experimental Algorithms*. Springer. 2012, pp. 369–380 (cit. on p. 29).

[43] Ilya Safro, Dorit Ron, and Achi Brandt. "Multilevel algorithms for linear ordering problems." In: *Journal of Experimental Algorithmics (JEA)* 13 (2009), p. 4 (cit. on p. 29).

[44] Cédric Chevalier and Ilya Safro. "Comparison of coarsening schemes for multilevel graph partitioning." In: *International Conference on Learning and Intelligent Optimization*. Springer. 2009, pp. 191–205 (cit. on p. 30).

[45] Alex Pothen, Horst D Simon, and Kang-Pu Liou. "Partitioning sparse matrices with eigenvectors of graphs." In: *SIAM journal on matrix analysis and applications* 11.3 (1990), pp. 430–452 (cit. on p. 30).

[46] Delel Rhouma and Lotfi Ben Romdhane. "An efficient multilevel scheme for coarsening large scale social networks." In: *Applied Intelligence* 48.10 (2018), pp. 3557–3576 (cit. on p. 30).

[47] Roland Glantz, Henning Meyerhenke, and Christian Schulz. "Tree-based coarsening and partitioning of complex networks." In: *International Symposium on Experimental Algorithms*. Springer. 2014, pp. 364–375 (cit. on p. 30).

[48] Daniel Delling, Andrew V Goldberg, Ilya Razenshteyn, and Renato F Werneck. "Graph partitioning with natural cuts." In: *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2011, pp. 1135–1146 (cit. on p. 30).

[49] Stephane Lafon and Ann B Lee. "Diffusion maps and coarse-graining: A unified framework for dimensionality reduction, graph partitioning, and data set parameterization." In: *IEEE transactions on pattern analysis and machine intelligence* 28.9 (2006), pp. 1393–1403 (cit. on p. 30).

[50] Cornelius Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA, 1950 (cit. on p. 30).

[51] George Karypis and Vipin Kumar. "A fast and high quality multilevel scheme for partitioning irregular graphs." In: *SIAM Journal on scientific Computing* 20.1 (1998), pp. 359–392 (cit. on pp. 28, 30).

[52] Roberto Battiti and Alan A. Bertossi. "Greedy, prohibition, and reactive heuristics for graph partitioning." In: *IEEE Transactions on Computers* 48.4 (1999), pp. 361–385 (cit. on pp. 30, 31).

[53] Brian W Kernighan and Shen Lin. "An efficient heuristic procedure for partitioning graphs." In: *Bell system technical journal* 49.2 (1970), pp. 291–307 (cit. on p. 31).

[54] Keld Helsgaun. "An effective implementation of the Lin–Kernighan traveling salesman heuristic." In: *European Journal of Operational Research* 126.1 (2000), pp. 106–130 (cit. on p. 32).

[55] Charles M Fiduccia and Robert M Mattheyses. "A linear-time heuristic for improving network partitions." In: *19th Design Automation Conference*. IEEE. 1982, pp. 175–181 (cit. on p. 32).

[56] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. "Engineering a Scalable High Quality Graph Partitioner." In: *arXiv preprint arXiv:0910.2004* (2009) (cit. on p. 33).

[57] Laura A Sanchis. "Multiple-way network partitioning." In: *IEEE Transactions on Computers* 38.1 (1989), pp. 62–81 (cit. on p. 33).

[58] Philippe Galinier, Zied Boujbel, and Michael Coutinho Fernandes. "An efficient memetic algorithm for the graph partitioning problem." In: *Annals of Operations Research* 191.1 (2011), pp. 1–22 (cit. on p. 34).

[59] Chris Walshaw. "Multilevel refinement for combinatorial optimisation problems." In: *Annals of Operations Research* 131.1-4 (2004), pp. 325–372 (cit. on p. 34).

[60] George Karypis, Kirk Schloegel, and Vipin Kumar. "Parmetis." In: *Parallel graph partitioning and sparse matrix ordering library. Version* 2 (2003) (cit. on pp. 35, 104, 137).

[61] Henning Meyerhenke, Peter Sanders, and Christian Schulz. "Parallel Graph Partitioning for Complex Networks." In: *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society. 2015, pp. 1055–1064 (cit. on p. 35).

[62] Cédric Chevalier and François Pellegrini. "PT-Scotch: A tool for efficient parallel graph ordering." In: *Parallel computing* 34.6-8 (2008), pp. 318–331 (cit. on p. 35).

[63] Kirk Schloegel, George Karypis, and Vipin Kumar. "Parallel multilevel algorithms for multi-constraint graph partitioning." In: *European Conference on Parallel Processing*. Springer. 2000, pp. 296–310 (cit. on pp. 35, 36, 39, 104, 137).

[64] George Karypis and Vipin Kumar. "Multilevelk-way partitioning scheme for irregular graphs." In: *Journal of Parallel and Distributed computing* 48.1 (1998), pp. 96–129 (cit. on pp. 27, 35, 36, 104, 137).

[65] George Karypis and Vipin Kumar. "A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm." In: *PPSC*. 1997 (cit. on pp. 35, 36, 104, 137).

[66] Marsha J Berger and Shahid H Bokhari. "A partitioning strategy for nonuniform problems on multiprocessors." In: *IEEE Transactions on Computers* 5 (1987), pp. 570–580 (cit. on p. 35).

[67] Andrew Y Ng, Michael I Jordan, and Yair Weiss. "On spectral clustering: Analysis and an algorithm." In: *Advances in neural information processing systems*. 2002, pp. 849–856 (cit. on pp. 44, 45, 62).

[68] Edward F Moore. "The shortest path through a maze." In: *Proc. Int. Symp. Switching Theory, 1959*. 1959, pp. 285–292 (cit. on p. 76).

[69] Zuse Konrad. "Der Plankalkül." In: *Konrad Zuse Internet Archive* (1972), pp. 96–105 (cit. on p. 76).

[70] Charles Pierre Trémaux. "École Polytechnique of Paris (X: 1876)." In: *Proc. French Engineer of the Telegraph in Public Conference*. 2010, pp. 1859–1882 (cit. on pp. 48, 76).

[71] Robert Sedgewick. *Algorithms in C++ Part 5: Graph Algoritms (Pt. 5)*. 2002 (cit. on pp. 48, 76).

[72] Shimon Even. *Graph algorithms*. Cambridge University Press, 2011 (cit. on pp. 48, 76).

[73] Donald E Knuth. "The art of computer programming, Vol. 1: Fundamental algorithms. 1968." In: *Vol II: Seminumerical algorithms* 3 (1969) (cit. on p. 160).

[74] Karen D Devine, Erik G Boman, Robert T Heaphy, Rob H Bisseling, and Umit V Catalyurek. "Parallel hypergraph partitioning for scientific computing." In: *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2006, 10–pp (cit. on p. 104).

[75] Erik G Boman, Ümit V Çatalyürek, Cédric Chevalier, and Karen D Devine. "The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering and coloring." In: *Scientific Programming* 20.2 (2012), pp. 129–150 (cit. on pp. 35, 104, 137).

[76] George Karypis and Vipin Kumar. "A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices." In: *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN* (1998) (cit. on p. 104).

[77] Andrea Romanoni and Matteo Matteucci. "Incremental reconstruction of urban environments by edge-points delaunay triangulation." In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2015, pp. 4473–4479 (cit. on pp. 154, 155).

[78] Andrea Romanoni and Matteo Matteucci. "Efficient moving point handling for incremental 3d manifold reconstruction." In: *International Conference on Image Analysis and Processing*. Springer. 2015, pp. 489–499 (cit. on pp. 69, 116, 154, 155).

[79] Andrea Romanoni, Amaël Delaunoy, Marc Pollefeys, and Matteo Matteucci. "Automatic 3d reconstruction of manifold meshes via delaunay triangulation and mesh sweeping." In: *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE. 2016, pp. 1–8 (cit. on pp. 69, 116, 154, 155).

[80] Andrea Romanoni, Marco Ciccone, Francesco Visin, and Matteo Matteucci. "Multi-view stereo with single-view semantic mesh refinement." In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 706–715 (cit. on pp. 69, 116).

[81] ANDREA ROMANONI. "Incremental large-scale visual 3D mesh reconstruction." PhD thesis. Italy, 2017 (cit. on pp. 154, 155).

[82] Hassan Alhaija, Siva Mustikovela, Lars Mescheder, Andreas Geiger, and Carsten Rother. "Augmented Reality Meets Computer Vision: Efficient Data Generation for Urban Driving Scenes." In: *International Journal of Computer Vision (IJCV)* (2018) (cit. on pp. v, 65).

[83] Jakob Engel, Thomas Schöps, and Daniel Cremers. "LSD-SLAM: Large-scale direct monocular SLAM." In: *European conference on computer vision*. Springer. 2014, pp. 834–849 (cit. on p. 153).

[84] Richard A Newcombe, Steven J Lovegrove, and Andrew J Davison. "DTAM: Dense tracking and mapping in real-time." In: *2011 international conference on computer vision*. IEEE. 2011, pp. 2320–2327 (cit. on p. 153).

[85] Pierre Moulon, Pascal Monasse, Romuald Perrot, and Renaud Marlet. "Openmvg: Open multiple view geometry." In: *International Workshop on Reproducible Research in Pattern Recognition*. Springer. 2016, pp. 60–74 (cit. on p. 154).

[86] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. "ORB-SLAM: a versatile and accurate monocular SLAM system." In: *IEEE transactions on robotics* 31.5 (2015), pp. 1147–1163 (cit. on p. 154).

[87] David Gallup, Jan-Michael Frahm, Philippos Mordohai, Qingxiong Yang, and Marc Pollefeys. "Real-time plane-sweeping stereo with multiple sweeping directions." In: *2007 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2007, pp. 1–8 (cit. on p. 155).