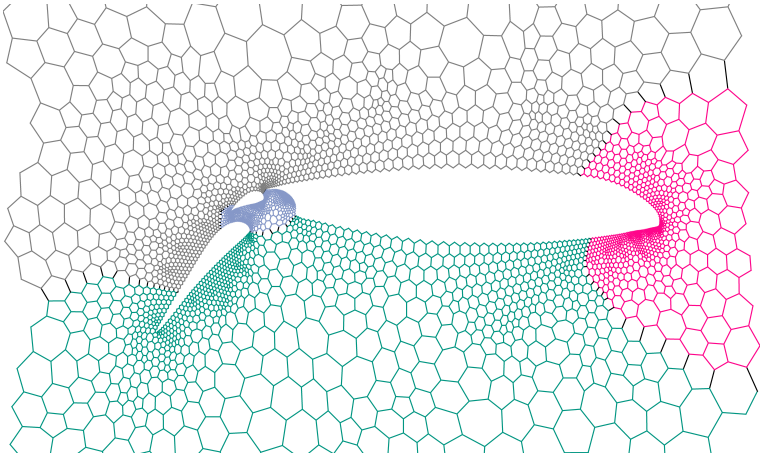
Course Notes:

Graph Partitioning and Graph Clustering in Theory and Practice

Christian Schulz

Institute for Theoretical Informatics
Karlsruhe Institute of Technology (KIT)

Revision: May 20, 2016



Involved Students: Sebastian Lamm, Henning Schulz, Demian Hespe, Eike Röhrs, Sebastian Korbinian Bayer, Christoph Hess, Christian Steiger, Robert Hangu, Marvin Teichmann, Jan Jacob, Fellipe Bernardes-Lima, Robert Hangu, Matthias Stumpp, Sergey Hayrapetyan

Preface

I worked on partitioning and clustering problems for almost seven years now. Yet, I still find these problems incredibly fascinating, there is so much work to be done and the field is evolving at such an high pace. I am really glad to be able to work on such interesting topics and hope that I can convince or at least transfer a little bit of my fascination on to *you* – the reader of this script and/or student of my lecture.

The course notes in front of you collect material needed to follow the course "Graph Partitioning and Graph Clustering in Theory and Practice" which has been first held at the Karlsruhe Institute of Technology (KIT) in the summer of 2014. This script has been mostly created by students attending the lecture in 2015. I am truly thankful to Sebastian Lamm, Henning Schulz, Demian Hespe, Eike Röhrs, Sebastian Korbinian Bayer, Christoph Hess, Christian Steiger, Robert Hangu, Marvin Teichmann, Jan Jacob, Fellipe Bernardes-Lima, Robert Hangu, Matthias Stumpp and Sergey Hayrapetyan for assembling the material as part of course homework. Hence, note that this is a script, not a text book. It is meant to accompany the lecture and not to replace it. In addition, although the course is in German, the following pages are written in English and *largely* compiled from papers, textbooks or dissertations written in English (using the original wordings). Hence, follow the citations to the original sources since there is little original material here. The state of the scriptum is very preliminary, so read the texts very careful and in case you find a mistake send me an email.

Karlsruhe, Mai 2015
Vienna, September 2015

Contents

Chapter 1

Introduction

1.1 Motivation

Graphs are frequently used by computer scientists as abstractions when modeling an application problem. Cutting a graph into smaller pieces is one of the fundamental algorithmic operations. Even if the final application concerns a different problem (such as traversal, finding paths, trees, and flows), partitioning or clustering large graphs is often an important subproblem for complexity reduction or parallelization. With the advent of even larger instances in applications such as scientific simulation, social networks, or road networks, *graph partitioning* and *graph clustering* therefore becomes highly important, multifaceted, and challenging.

Probably the most-known application of graph partitioning is to balance load and minimize communication in scientific simulations [120, 30, 48]. Other applications speed up Dijkstra's algorithm [89, 98] or algorithms in the route planning area [91, 82, 38]; Graph clustering on the other hand, aims at revealing structures present in graphs with applications ranging from the field of social sciences of biology to the growing field of complex systems. We present some selected applications of graph partitioning and clustering in the foregoing chapter.

What makes these problems even more appealing is the fact most of them are NP-complete and not easy to approximate. Hence, mostly heuristics are used in practice. These heuristics usually use an astonishingly large set of "easier" graph algorithms to tackle the problem. For example, algorithms such as weighted matching, spanning trees, edge coloring, breadth-first search, dominating sets, maximum flows, diffusion, negative cycle detection, shortest paths, and strongly connected components as well as lots of very interesting data structures. We discuss some of these problems within the lecture. A commonly used method to partition or cluster large graphs is the *multilevel approach* or variations thereof. Here, the graph is recursively *contracted* to create smaller graphs which somewhat

should reflect the same basic structure as the input graph. This usually achieved by modifying edge and node weights of the coarser graphs. Often the weight of a coarse node is the number of the contracted nodes that it represents and the weight of an edge stands for the number of parallel edges that it replaces. This way, a solution of a coarse level creates a solution of the finer graphs. After applying an *initial* algorithm to solve the problem on the smallest graph, the contraction is undone and, at each level, a *local search* method may be used to improve the solution induced by the coarser level. As we will see later, in some graph clustering cases local search can be omitted and the output of the algorithm is the computed hierarchy itself. The intuition behind this approach is that a good solution at one level of the hierarchy will also be a good solution on the next finer level so that local search converges quickly, i.e. rapidly finds a good solution. On the other hand, local search has a somewhat global view on the optimization problem on the coarse levels of the multilevel approach, whereas it has very fine view on the fine levels of the hierarchy. We will discover later on that the global view in the graph partitioning case stems from the fact that the movement of a node on the coarse level represents the movement of a whole set of nodes on the finest level of the hierarchy.

1.2 Lecture Overview

Throughout the lecture we cover different aspects of graph partitioning and graph clustering. After defining these problems and its objective functions properly, we show the hardness of some variants and present algorithms having a global view such as integer linear programs or spectral techniques. A large amount of the lecture is centered around the multilevel scheme which is explained in high detail for the graph partitioning problem. We then cover parallel, (semi-)external and evolutionary algorithms to tackle the problems. In addition, we talk about the currently most important algorithms to cluster graphs effectively. As mentioned above, most of the algorithms make use of “easier” graph algorithms/problems which will be explained at the point of occurrence.

1.3 Fundamentals / Review of Linear Algebra

We now present fundamental and notation that is used throughout the script.

General Graph Related Definitions. A weighted (directed) *graph* G consists of a set of nodes V and a set of edges $E \subset V \times V$ to represent relations between the nodes, as well as two cost functions. One function assigns weights to the nodes $c : V \to \mathbb{R}_{>0}$ and a second function $\omega : E \to \mathbb{R}$ assigns costs to the edges. In general, we write n for the number of nodes and m for the number of edges. In an undirected graph an edge $(u, v) \in E$ implies an edge $(v, u) \in E$ and that both edge weights are equal. We use the set notation $\{u, v\} \in E$ in the undirected case. We extend c and ω to sets, i.e. $c(V')$:= $\sum_{v \in V'} c(v)$ and $\omega(E')$:= $\sum_{e \in E'} \omega(e)$. The set $\Gamma(u)$:= $\{v : \{u, v\} \in E\}$ denotes the neighbors of a node u. The *degree* $d(v)$ of a node v is the number of its neighbors. With Δ we denote the *maximum degree* of a graph. The weighted degree of a node is the sum of the weights of its incident edges. A graph is *bipartite* if its node set can be divided into two disjoint sets U and V such that $\{u, v\} \in E$ implies $u \in U$ and $v \in V$ or vice versa. A *subgraph* is a graph whose node and edge set are subsets of another graph. We call a subgraph induced if it has every possible edge.

Flows and Cuts. In a directed graph with two designated nodes, a source s and a sink t, that has *non-negative* edge weights (serving as capacities), a *flow* is a function $f : V \times V \to \mathbb{R}$ that satisfies a capacity constraint, a flow conservation constraint and a skew-symmetry constraint. The *capacity constraint* demands that the flow value associated is lower or equal to the capacity ($f(u, v) \leq \omega(u, v)$) and the *flow conservation constraint* requires that each node emits the same amount of flow as it receives – except the two designated nodes source s and sink t. The *skew symmetry constraint* requests $f(u, v) = -f(v, u)\ \forall (u, v) \in V \times V$. The value $f(u, v)$ denotes the amount of flow on an edge (u, v). The value of a flow val(f) is defined as the total amount of flow that is transferred from the source to the sink. The *residual capacity* is defined as $r_f(u, v) = \omega(u, v) - f(u, v)$. The *residual graph* for a directed graph $G = (V, E)$ and a flow f is given as $G_f = (V, E_f)$ where $E_f = \{(u, v) \in V \times V \mid r_f(u, v) > 0$ and $(u, v) \in E$ or $(v, u) \in E\}$.

An s-t *cut* is defined as tuple $(S, V \setminus S)$ with $s \in S \subset V$ and $t \in V \setminus S$. The weight of an s-t cut is defined as $\sum_{(u,v) \in E \cap S \times V \setminus S} \omega(u, v)$, i.e. the weight of the edges starting in S and ending in $V \setminus S$. A minimum s-t cut has the smallest weight among all s-t cuts. It is well-known that the value of a maximum s-t flow corresponds to the value of a minimum s-t cut, and if a maximum s-t flow is given, then a minimum cut is easily computed.

Further Notation and Definitions. A *matching* $M \subseteq E$ is a set of edges that do not share any common nodes, i.e. the graph (V, M) has maximum degree one. The weight of a matching is defined as the weight induced by its edges $\omega(M)$. A matching is said to be *maximal* if there is no edge that can be added to the matching, and a matching that has maximum weight among all matchings is called a *maximum weight matching*. For a graph a subset $C \subseteq V$ is a *closed node set* if and only if for all nodes $u, v \in V$, the conditions $u \in C$ and $(u, v) \in E$ imply $v \in C$. In other words, a subset C is a *closed node set* if there is no edge starting in C and ending in its complement $V \setminus C$. A subset $D \subseteq V$ is a *dominating set* if for each $v \in V$ either v itself or one of its neighbors is contained in D. D is called an *independent set* if the nodes of D don't share an edge.

A sequence of nodes $s \to \cdots \to t$ such that each pair of consecutive nodes is connected by an edge, is called an *s-t path*. We say that s is the source and t is the target. The length of a path is defined by the sum of its edge weights. A shortest s-t path is a path with the smallest weight among all s-t paths. A path with equal source and target is called a *cycle*. It is *simple* if no node is contained twice. A cycle with negative weight is also called *negative cycle*. A directed graph is *strongly connected* if there is a u-v path and a v-u path for each pair of nodes u, v. A maximal strongly connected induced subgraph is called *strongly connected component*. These concepts are transferred straightforwardly to undirected graphs. However, the term *connected component* is used instead. In a directed graph, a linear ordering $\prec$ of the nodes such that an edge $(u, v) \in E$ implies that $u \prec v$ in the ordering is called a *topological order*.

1.4 Graph Partitioning

We are now ready to define the graph partitioning problem. Given a number $k \in \mathbb{N}_{>1}$ and an undirected graph with *non-negative* edge weights, the *graph partitioning problem* asks for *blocks* of nodes $V_1,\ldots,V_k$ that partition the node set V, i.e.

$$1.\ V_1 \cup \cdots \cup V_k = V$$
$$2.\ V_i \cap V_j = \emptyset\ \ \forall i \neq j.$$

A *balance constraint* demands that all blocks have about equal size. More precisely, it requires that,

$$\forall i \in \{1..k\} : |V_i| \leq L_{\max} := (1+\epsilon)\lceil |V|/k \rceil$$

for some imbalance parameter $\epsilon \in \mathbb{R}_{\geq 0}$ in the case that the cost function of the nodes is identical to one. Often after one has computed a partition of a graph one reports the *maximum imbalance* $\max_i |V_i|/\lceil |V|/k \rceil$ of the partition.

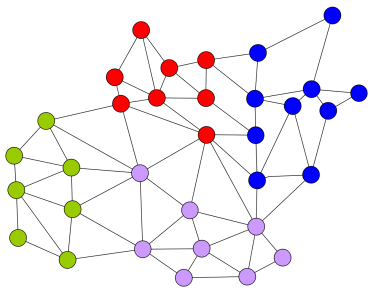


Figure 1.1: An example graph that is partitioned into four blocks. The partition has 17 cut edges. The weights of the blocks are blue(10), red(8), green(7) and purple(8). Hence, the maximum imbalance is 11%.

Notation. In the case of $\epsilon = 0$, we also use the term *perfectly balanced*. A block V_i is *underloaded* if $|V_i| < L_{\max}$ and *overloaded* if $|V_i| > L_{\max}$. A node $v \in V_i$ that has a neighbor $w \in V_j$, $i \neq j$, is a *boundary node*. An edge that runs between blocks is also called *cut edge*. The set E_{ij}:= $\{\{u, v\} \in E : u \in V_i, v \in V_j\}$ is the set of cut edges between two blocks V_i and V_j. An abstract view of the partitioned graph is the so called *quotient graph*, where nodes represent blocks and edges are induced by connectivity between blocks, i.e. there is an edge in the quotient graph if there is an edge that runs between the blocks in the original, partitioned graph. An example is given in Figure 1.2.

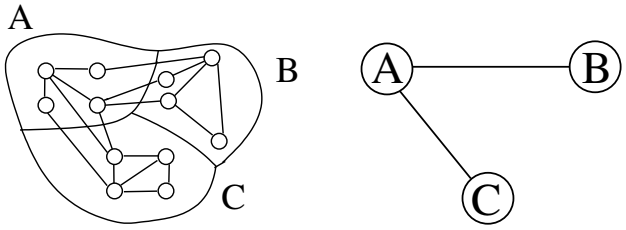


Figure 1.2: A graph that is partitioned into three blocks of size four on the left and its corresponding quotient graph on the right. There is an edge in the quotient graph if there is an edge between the corresponding blocks in the original graph.

1.4.1 Objective Functions

In practice, we often seek to find a partition that minimizes (or maximizes) an objective. Probably the most prominent objective function is to minimize the *total cut*

$$\sum_{i<j} \omega(E_{ij}).$$

In other words, the objective computes the sum of the weight of the cut edges. It is well-known that there are more realistic (and more complicated) objective functions involving also the block that is worst and the number of its neighboring nodes [60] which we will also partially discuss. It is worth mentioning that during the last decades minimizing the cut size has been adopted as a kind of standard, since it is usually highly correlated with the other formulations on mesh type networks. These networks yielded (and still do) one of the main applications of graph partitioning. However, the standard may be changing in the future since nowadays applications that need to partition social networks or web graphs are emerging. On this kind of networks different objective functions are often not that correlated.

Lets discuss this in a little bit more detail. Again, for applications where models of computation and communication need to be partitioned there are more accurate objective functions measuring the communication volume. These objectives measure the communication that is done in the system. A typical example of such an application is shown in Figure 1.3. In such applications the graph usually needs to be distributed across several processors and communication between processors takes place for nodes that have non-local edges. Note the blue vertex in the example: the cut measures two units of communication whereas a real application will only send the information attached to the vertex once to processor

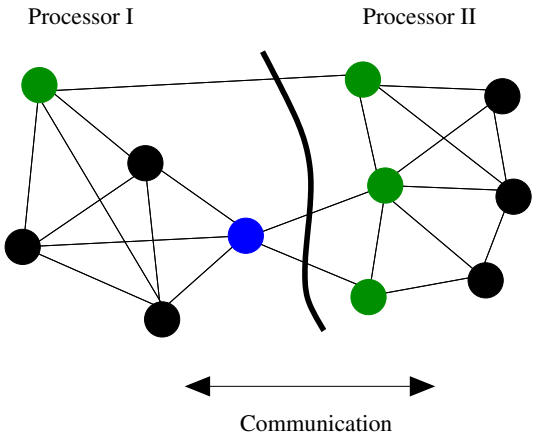


Figure 1.3: A typical model of computation and communication.

II. Hence, there are objectives measuring the total or the maximum communication volume. For a block V_i, the communication volume is defined as $\text{comm}(V_i) := \sum_{v \in V_i} c(v)D(v)$, where $D(v)$ denotes the number of different blocks in which v has a neighbor node, excluding V_i. The *maximum communication volume* is then defined as $\max_i \text{comm}(V_i)$ and the *total communication volume*, sums over the communication volumes of the blocks. The maximum objective can be motivated by the fact that in parallel applications one usually has to wait for the processor that finishes a communication step last.

Other versions of the graph *bi*partitioning problem do not have a fixed balance constraint but encode balance directly in the objective function. For example, the *expansion* of a non-trivial cut (V_1, V_2) is defined as

$$\frac{\omega(E_{12})}{\min(c(V_1), c(V_2))}.$$

Similarly, the *conductance* of such a cut is defined as

$$\frac{\omega(E_{12})}{\min(\text{vol}(V_1), \text{vol}(V_2))},$$

where $\text{vol}(S) := \sum_{v \in S} d(v)$ denotes the volume of the set S.

1.4.2 Common Applications of Graph Partitioning

We now discuss some exemplary applications of graph partitioning.

Finite Element Method / Scientific Computing. This is one of the main applications for graph partitioning. In our example, we very briefly outline the finite element method (leaving out most of the details) and then see how we can use graph partitioning techniques to solve the arising linear equation systems on a parallel computer. One advantage of the finite element method is that the corresponding linear equation systems are sparse, e.g. they have only $O(n)$ or $O(n \log n)$ non-zero

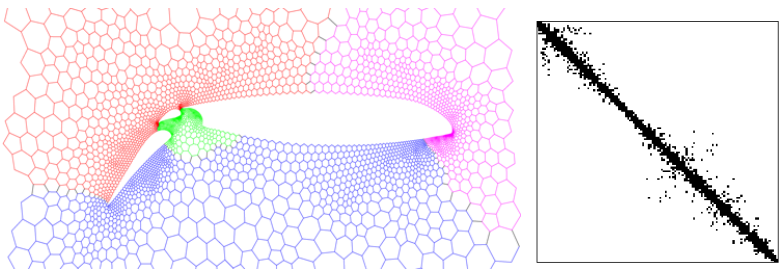


Figure 1.4: An example mesh around an airfoil and its linear equation system.

elements. In scientific simulations one often wants to compute an approximation to some partial differential equation that models a physical process. In the following example we use the heat equation for that purpose. Let Ω be a region.

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega \\ u &= 0 \text{ on } \partial\Omega \end{aligned}$$

with $\Delta = \sum \partial_{ii}$. We want to find a function $u \in L_2(\Omega)$ that solves the above system. However, the solution space does not have finite dimension and deriving an explicit formula for u is rarely possible. Hence, we an approximation to the system. One possibility to do so is to use so called weak formulations of the system: Find $u \in V$ such that

$$a(u, v) = (f, v)_{L^2(\Omega)} \; \forall v \in V$$

where $a(u, v) := \int_\Omega \nabla u \cdot \nabla v dx$ a contiguous bi-linear map. In a second step, the solution space is approximated by a vector space of finite dimension, e.g. choosing $V_h := \text{span}\{\phi_1, \cdots, \phi_n\} \subset V$. Our weak formulation now becomes: Find $u_h \in V_h$ such that

$$a(u_h, \phi) = (f, \phi)_{L^2(\Omega)} \; \forall \phi \in V_h.$$

We now do a couple of transformation to get a linear equation system. First, since V_h has finite dimension we have $\exists \alpha_i \in \mathbb{R}$ such that $u_h = \sum_{i=1}^n \alpha_i \phi_i$. Hence, we want to find α_i such that

$$\begin{aligned} & a(\sum_{i=1}^n \alpha_i \phi_i, \phi) \; = (f, \phi)_{L^2(\Omega)} \; \forall \phi \in V_h \\ \Leftrightarrow & \sum_{i=1}^n \alpha_i a(\phi_i, \phi_j) = (f, \phi_j)_{L^2(\Omega)} \; \forall j \in \{1, \cdots, n\} \\ \Leftrightarrow & Ax = b \end{aligned}$$

with

$$A = \begin{pmatrix} a(\phi_1, \phi_1) & \dots & a(\phi_1, \phi_n) \\ \vdots & & \vdots \\ a(\phi_n, \phi_1) & \dots & a(\phi_n, \phi_n) \end{pmatrix}, \quad b = \begin{pmatrix} (f, \phi_1) \\ \vdots \\ (f, \phi_n) \end{pmatrix}$$

Now for reasons that are not the focus of the lecture, most of the entries of the matrix are zero. Hence, the linear equation system is mostly solved using an iterative method such as,e.g. the Jacobi method which requires sparse matrix vector multiplication as a core component. In essence, at each time step we have an approximate solution x_k to the linear equation system and computes a new approximation x^{k+1} using the equation

$$x^{k+1} = D^{-1}(b - Rx^k)$$

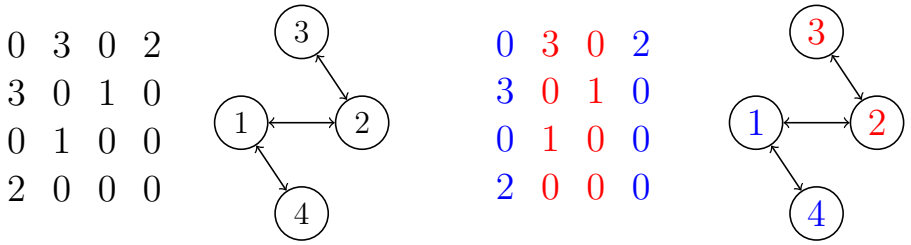


Figure 1.5: An example matrix and its corresponding graph model of computation and communication. As well as its partitioned version.

where $D = diag(A)$, $R = A - D$, and repeats this until the process converged. In order to parallelize this we have to distribute columns of A and x over equally processors such that communication is minimized. To build our model of computation and communication we rewrite the iterative formula.

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{i \neq j} a_{ij} x_j^k\right)$$

We define a graph $G = (V = \{1, \cdots, n\}, E)$ which has node per column of the matrix and edges such that $e = \{i, j\} \in E \Leftrightarrow a_{i,j} \neq 0$. Additionally, we associate $x_i^k, x_i^{k+1}, A_{i,*}$ with node $i \in V$. We can now partition the graph. PE l obtains columns associated with V_l, stores $x_i^k, x_i^{k+1}, A_{i,*}$ and computes x_i^{k+1} for $i \in V_l$. Hence, a partition distributed the work load and minimizes communication. Since, the work load to be done for a node is proportional to its degree, we can use the degree as vertex weight in our model to account for that.

Customizable Route Planning. The goal of the customizable route planning (CRP) problem is to perform real-time queries on road networks with arbitrary metrics. Such algorithms can be used in two scenarios: they may keep several active metrics at once (to answer queries for any of them), or new metrics can be generated on the fly. A system with these properties has obvious attractions. It supports real-time traffic updates and other dynamic scenarios, allows easy customization by handling any combination of standard metrics, and can even provide personalized driving directions (for example, for a truck with height and weight restrictions). To implement such a system, one needs an algorithm that allows real-time queries, has fast customization (a few seconds), and keeps very little data for each metric. Most importantly, it must be robust: all three properties must hold for any metric. To achieve these goals, CRP distinguished between two features of road networks. The topology is a set of static properties of each road segment or turn, such as physical length, road category, speed limits, and turn types. The metric encodes the actual cost of traversing a road segment or taking a turn. It can often be described compactly, as a function that maps (in constant time) the

properties of an edge/turn into a cost. It is assumed that the topology is shared by the metrics and rarely changes, while metrics may change quite often and even coexist. To exploit this separation, algorithms having three stages are considered for customizable route planning. The first, metric-independent preprocessing, may be relatively slow, since it is run infrequently. It takes only the graph topology as input, and may produce a fair amount of auxiliary data (comparable to the input size) – this is where graph partitioning will be used. The second stage, metric customization, is run once for each metric, and must be much quicker (a few seconds) and produce little data—a small fraction of the original graph. Finally, the query stage uses the outputs of the first two stages and must be fast enough for real-time applications

Basic Algorithm. The metric-independent preprocessing stage partitions the graph into connected blocks with at most U (an input parameter) vertices each, with as few cut edges (edges with endpoints in different blocks) as possible. The metric customization stage builds a graph H containing all boundary vertices (those with at least one neighbor in another block) and cut edges of G. It also contains a clique for each block C: for every pair (v, w) of boundary vertices in C, an edge (v, w) is created whose cost is the same as the shortest path (restricted to C) between v and w (or infinite if w is not reachable from v). This is done by running Dijkstra from each boundary vertex. Note that H is an overlay [123]: the distance between any two vertices in H is the same as in G. Finally, to perform a query between s and t, a bidirectional version of Dijkstra's algorithm on the graph consisting of the union of H, C_s , and C_t is executed. Here, C_v denotes the subgraph of G induced by the vertices in the block containing v.

Influence of Graph Partitioning. There is lots of engineering work in the original paper [38] that make the methods highly competitive. Additionally, recent advances in graph partitioning had a large influence to the success of the method. In particular, HiTi [73] uses similar techniques based on edge separators and cliques to represent each block. Unfortunately, HiTi has not been tested on large road networks and does not use high quality graph partitioning techniques. CRP, drastically improved query speedups relative to Dijkstra from less than 60 [67] to more than 3000. This makes real-time queries possible. Furthermore, by explicitly separating metric customization from graph partitioning, new metrics are enabled to be processed in a few seconds. The result is a flexible and practical solution to many real-life variants of the problem. It should be straightforward to adapt it to augmented scenarios, such as mobile or time-dependent implementations. (In particular, a unidirectional version of MLD is also practical.) Since partitions have a direct effect on performance, it may also be interesting to improve them further, perhaps by explicitly taking the size of the overlay graph into account.

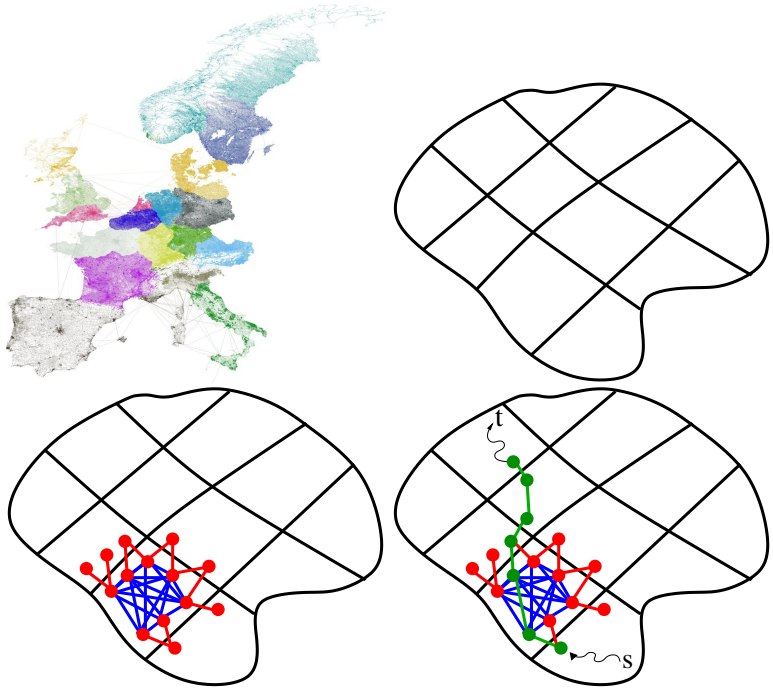


Figure 1.6: Illustrating CRP. The original network is partitioned. An overlay network is computed. $s-t$ queries are done on the graph that consists of the union of H, C_s and C_t.

1.5 Graph Clustering

A *clustering* is also a partition of the nodes, however k is usually not given in advance and the balance constraint is removed. Note that a partition is also a clustering of a graph. In both cases, the *goal* is to minimize or maximize a particular objective function. Sometimes, there is an upper bound to the cluster sizes specified. One of the main paradigms of clustering is to find groups/clusters intra-cluster density vs. inter-cluster sparsity. As we will see this can be formalized with a number of objective functions.

Notation. Given an unweighted graph $G = (V, E)$. Let $\mathcal{C} = \{C_1, ..., C_k\}$ be a partition of V, with each C_i being non-empty. We call $\mathcal{C}$ a *clustering* of G and the elements C_i *clusters*. The cluster which contains node v is denoted by $\mathcal{C}(v)$. We identify a cluster C_i with the node-induced subgraph of G, i.e., the graph $G[C_i] := (C_i, E(C_i), \omega_{|E(C_i)})$, where $E(C_i) := \{\{v, w\} \in E : v, w \in C_i\}$. Then $E(\mathcal{C}) := \bigcup_{i=1}^{k} E(C_i)$ is the set of *intra-cluster edges* and $E \backslash E(\mathcal{C})$ the set of *inter-cluster edges*, with $|E(\mathcal{C})| =: m(\mathcal{C})$ and $|E \backslash E(\mathcal{C})| =: \overline{m}(\mathcal{C})$. The set $E(C_i, C_j)$ denotes the set of edges connecting nodes in C_i to nodes in C_j. We

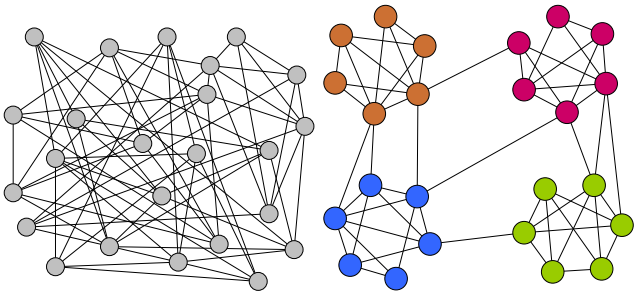


Figure 1.7: An example clustering of a graph.

denote the number of non-adjacent intra-cluster pairs of nodes as $m(\mathcal{C})^c$, and the number of non-adjacent inter-cluster pairs as $\overline{m}(C)^c$. Further, we generalize degree $deg(v)$ to clusters as $deg(C) := \sum_{v \in C} deg(v)$. A clustering is called *trivial* if either $k = 1$, or all clusters contain only one element, i.e., are singletons. When using edge weights, all the above definitions generalize naturally by using $\omega(e)$ instead of 1 when counting edge e. For the purpose of clustering, weights are considered to represent similarities unless otherwise noted. Consider a weighted graph $G = (V, E, \omega)$, then $\omega(\mathcal{C})$ ($\overline{\omega}(\mathcal{C})$) denotes the sum of the weights of all intra-cluster (inter-cluster) edges, W denotes the sum of all edge weights. The maximum edge weight in a graph is called $\omega_{\max}$.

Given two clusterings $\mathcal{C}_1$ and $\mathcal{C}_2$, the *overlay clustering* is the clustering where each block corresponds to a connected component of the graph $G_{\mathcal{E}} = (V, E \backslash \mathcal{E})$ where $\mathcal{E}$ is the union of the cut edges of $\mathcal{C}_1$ and $\mathcal{C}_2$, i.e. all edges that run between blocks in either $\mathcal{C}_1$ or $\mathcal{C}_2$.

Objective Functions

We now cover the objective functions that are commonly used in the field. Before we start we present an desired properties of an objective function and recap an impossibility theorem.

Postulations to a Measure and Impossibility Theorem. Given a graph G and a clustering $\mathcal{C}$ a quality measure should behave as follows: more intra-edges $\Rightarrow$ higher quality, less inter-edges $\Rightarrow$ higher quality, clusters must be connected, random clusterings should have bad quality, disjoint cliques should approach maximum quality, locality of the measure, comparable results across instances, fulfill the desiderata of the application .... However, there is the following impossibility theorem [84]:

Theorem 1
Given set S. Let $f : d \mapsto \Gamma$ be a function on a distance function d on set S, returning a partition Γ. No function f can simultaneously fulfill the following:

- *Scale-Invariance*
  for any distance function d and any $\alpha > 0$, we have $f(d) = f(\alpha \cdot d)$
- *Richness*
  for any given partition of a set and any given value Γ, we should be able to define a distance function d such that $f(d) = \Gamma$
- *Consistency*
  if we build d' from d by reducing intra-distances and increasing inter-distances, we should have $f(d') = f(d)$

Coverage. The most simple index realizing a traditional measure of clustering quality is coverage. The coverage($\mathcal{C}$) of a graph clustering $\mathcal{C}$ is defined as the fraction of intra-cluster edges (or $\omega(C)$) within the complete set of edges (or W):

$$\text{cov}(C) := \frac{m(\mathcal{C})}{m}$$

$$\text{cov}_\omega := \frac{\omega(\mathcal{C})}{W}$$

Intuitively, large values of coverage correspond to a good quality of a clustering. However, one principal drawback of coverage is, that the converse is not necessarily true: Coverage takes its largest value of 1 in the trivial case where there is only one cluster. Finding a clustering with $k \geq 3$ with optimal coverage is equivalent to finding a k-min-cut which is NP-hard.

Performance. The index performance partly remedies the main drawback of coverage. It is defined as the fraction of node pairs, that are clustered correctly, i.e. those connected node pairs that are in the same cluster and those non-connected node pairs that are separated by the clustering. The unweighted case yields:

$$performance(\mathcal{C}) := \frac{m(\mathcal{C}) + \overline{m}^c(\mathcal{C})}{\frac{1}{2}n(n-1)}$$

The range of unweighted performance is [0, 1] for simple graphs. Loops do not change things, but parallel edges do. Maximizing performance is NP-hard [124] as it is reducible to graph partitioning. The drawback of performance is that in sparse networks, which most real-world networks indeed are, the value of $\overline{m}^c(\mathcal{C})$ clearly dominates the formula, supporting rather fine clusterings.

Inter-Cluster Conductance. Inter-cluster conductance (or inter-cc) measures the worst bottleneck constituted by cutting off a cluster from the graph, normalized by the degree sums thereby cut off. Inter-cc is based on the measure conductance [74], which seeks the "cheapest" cut $(S, V\backslash S)$ (with $S \subseteq V$) in a graph (measured by ϕ, the fractional term of Equation 1.1). The conductance of a clustering is then defined as the minimum conductance of each cluster. However, determining the minimum conductance cut in a graph is NP-hard [8], and thus this measure is ill-suited for measuring clustering quality. In turn, the cut induced by a cluster should have a very low conductance in a good clustering. Following [24] we can thus examine how good bottlenecks induced by clusters are (instead of all cuts inside a cluster), which yields the meaningful (and computable) formula given in Equation 1.1. We shape this measure such that it yields 1 for good clusterings. For brevity we only give a weighted formula, which can canonically be used for unweighted graphs ($\omega(v) \to deg(v), \omega(C) \to |E(C)|$):

$$\text{icc}_\omega(\mathcal{C}) := 1 - \max_{C \in \mathcal{C}} \frac{\omega(E(C, V\backslash C))}{\min \sum_{v \in C} \omega(v), \sum_{v \in V\backslash C} \omega(v)} \tag{1.1}$$

Inter-cc is a worst-case measure, and this fact should be kept in mind. Thus, a clustering has a small inter-cc, if there exists at least one cluster C_0, that is rather strongly connected to $V\backslash C_0$, compared to the density of C_0 and $V\backslash C_0$. A non-isolated singleton cluster immediately results in a value of 0. The range of inter-cluster conductance is always [0, 1], even for unweighted and non-simple graphs. We refer the reader to [24] for a discussion on why not to use intra-cluster conductance, which is the analog to inter-cc inside clusters. For clusterings of large real-world graphs it is rather common that inter-cc equals 0, due to some small degeneracy that may occur someplace. For this reason one can also define the less capricious measure average inter-cc, in order to still have a meaningful measure of bottlenecks.

Modularity. Modularity has recently been proposed [101] in an attempt to find an apt remedy to the disadvantages of coverage. Citing the authors, the driving idea for modularity was to take coverage "minus the expected value of the same quantity in a network with the same community divisions, but random connections between the vertices.". The commonly used formula is as follows:

$$\text{mod}(\mathcal{C}) := \textit{cov}(\mathcal{C}) - \mathbb{E}[\textit{cov}(\mathcal{C})] = \frac{m(\mathcal{C})}{m} - \frac{1}{4m^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} \text{deg}(v) \right)^2$$

The original formulation did not take into account loops and miscounted coverage for non-simple graphs. The literature has seen quite a few obfuscated or

straight-out wrong formulations for modularity, due to the above issue or some otherwise sloppy enumeration. Since originally (and by a few follow-up studies) loops and parallel edges were disregarded, the original definitions are inconsistent, if such were allowed. We omit the generalization of modularity to weighted edges. The formula of modularity reveals an inherent trade-off: To maximize the first term, many edges should be contained in clusters, whereas the minimization of the second term is achieved by splitting the graph into many clusters with small total degrees each, or at least with a rather balanced total degree. The range of modularity is [-0.5, 1] for all graphs, where the least values are attained by bipartite graphs (and the obvious clustering) and 1 is approached by disjoint cliques.

Probability Model. We did not yet define the probability model that we look at to compute the expected value of coverage. Intuitively, the model looks at random graphs that keep the *expected* node degrees and randomly throws in edges. The edge set is a multi-set. The clustering is kept. More precisely:

1. start with set V
2. keep expected degrees
3. edge attaches to node v with $p = \frac{\deg(v)}{2m}$
4. other end attaches to w with $p = \frac{\deg(w)}{2m}$

$\rightarrow\ \mathbb{P}[e = (v, w)] = \frac{\deg(v) \cdot \deg(w)}{4m^2}$

$\rightarrow\ \mathbb{P}[e = \{v, w\}] = \frac{\deg(v) \cdot \deg(w)}{2m^2}$

$\rightarrow\ \mathbb{E}[\mathrm{cov}(\mathcal{C})] = \frac{1}{4m^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} \deg(v) \right)^2$

$\rightarrow\ \mathrm{mod}(\mathcal{C}) = \frac{m(\mathcal{C})}{m} - \frac{1}{4m^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} \deg(v) \right)^2$

Discussion/Miscanellea. Modularity is easy to use and implement. It has reasonable behavior on many practical instances and hence is heavily used in various fields. The clusterings with high modularity are close to human intuition of quality. However, Modularity also has some drawbacks. For example it has a resolution limit which means that large graphs tend to have high modularity even if the clustering is not that good. Moreover, it would may becomes necessary to define application specific null/random models. Finding clusterings with maximum modularity is NP-hard even when it is restricted to case $|\mathcal{C}| = 2$. Greedy maximization does not approximate modularity. There are ILP-formulations for Modularity feasible for graphs with at most $\approx |V| \leq 200$ (we will cover them later).

Surprise. An even more recent metric is Surprise [7]. Given a clustering $\mathcal{C}$ of G – $V_1, \ldots, V_\ell$ – Surprise measures the probability that a random graph $\mathcal{R}$ has more intricate edges:

$$S(\mathcal{C}) = \mathbb{P}(m_{\mathcal{R}}(\mathcal{C}) \geq m_G(\mathcal{C}))$$

The random model are now all graphs labeled with V and exactly m edges. Intuitively, the smaller $S(\mathcal{C})$, the more *surprising* are intra-cluster edges. We now derive a practical formula to compute the Surprise of a clustering. The total number of possible pairs is $p = n(n-1)/2$ and the maximum number of pairs within clusters $M = \sum |V_i|(|V_i| - 1)/2$. We now consider an urn model (without replacement): it has M white balls (possible edges within cluster) and $p - M$ black balls (possible edges between cluster). This model has a hyper-geometric distribution. Hence:

$$S(\mathcal{C}) = \sum_{i=m_G(\mathcal{C})}^{m} \frac{\binom{M}{i}\binom{p-M}{m-i}}{\binom{p}{m}}$$

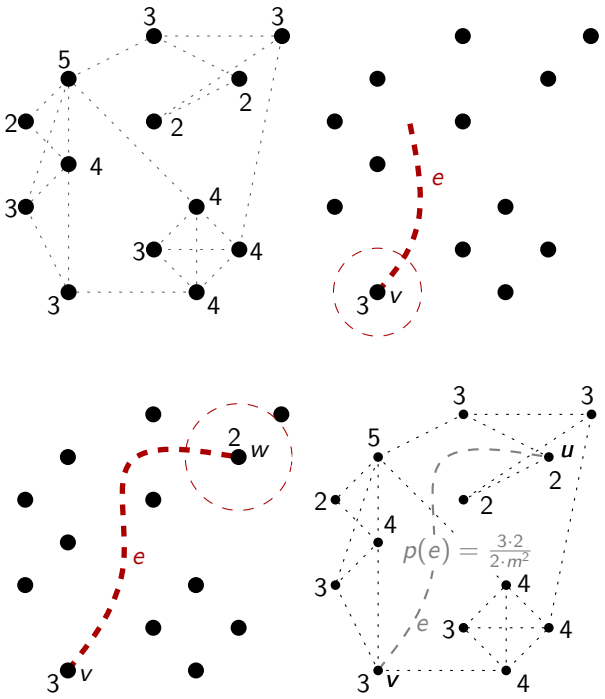


Figure 1.8: Probability model of modularity illustrated.

1.6 Clustering vs. Partitioning

	clustering	partitioning
purpose	analysis	handling instances
. . . and then?	zoom/abstraction	work on blocks
# of blocks	open	predefined
size of blocks	open	upper bound
criteria	various	various
constraints	often none	single, multiple

References

This chapter is largely based on Clauset et al. [1], Blondel et al. [20], and Flake et al. [50]. This chapter was created by Fellipe Bernardes-Lima.

Chapter 2

Some NP-hardness Results

This chapter presents important graph partitioning and graph clustering related NP-hardness results. We take a closer look at the NP-completeness of balanced bipartition, will see that there is no polynomial time constant factor approximation algorithm for perfectly balanced partitioning with $k \geq 3$ blocks unless P=NP, and show that conductance clustering is NP-complete as well. All of these problems yield important results for the computational complexity of graph partitioning and graph clustering.

Preliminaries

This section introduces necessary notations and definitions for this chapter.

P and NP. *P* and *NP* are complexity classes used for classifying decision problems. A *decision problem* can be defined as an algorithm that decides whether or not a certain input element is part of a given set M. An example of a decision problem is the Hamilton cycle problem. This problem determines if a given graph belongs to the set of graphs that contain a cycle that visits each vertex exactly once.

The class *P* describes the set of all decision problems that can be solved in time $O(n^d)$. Here, n describes the number of bits used for representing the input and d is a constant. Linear Programming and finding a maximum matching in a given graph are exemplary problems of this class.

NP is the class of decision problems for which the instances, where the answer is "yes", have efficiently verifiable proofs that the answer is yes. Efficiently verifiable means that they take polynomial time in the input size. In contrast, instances where the answer is "no" do not need to be efficiently verifiable. The class of problems that have this characteristic is called *co-NP*. Examples for problems in *NP* are the aforementioned Hamilton cycle problem and the traveling salesman problem.

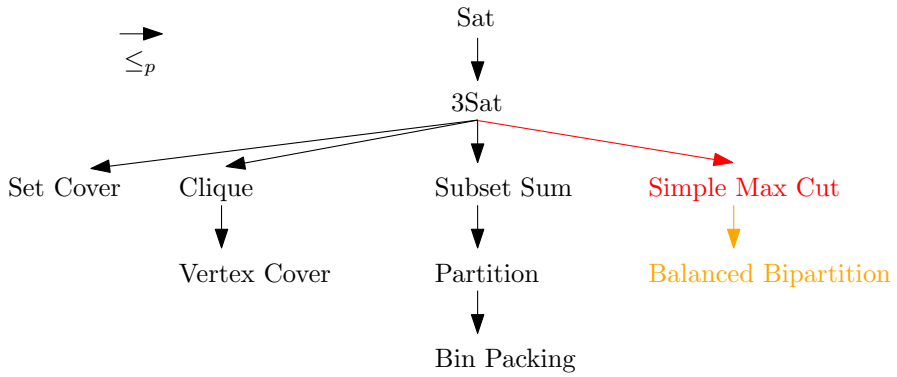


Figure 2.1: NP-hard problems and their according polynomial reductions.

NP-hardness. Let $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$ be languages over the alphabets Σ and Γ. A is called polynomial reducible to B (written $A \leq_p B$) if there exists a function $f : \Sigma^* \to \Gamma^*$ such that for all words $w \in \Sigma^*$ holds: $w \in A$ if and only if $f(w) \in B$. In addition to that, the function f has to be computable in polynomial time. Polynomial reducibility can be used to prove that certain problems are part of P or NP. For example, if for a given problem A we find another problem B that is in P and we can show that $A \leq_p B$ then this implies that $A \in P$. The same holds true if B is in NP. Another use of polynomial reducibility is to define the complexity classes NP-hard and NP-complete.

A problem A is *NP-hard* if every problem $L \in$ NP is polynomial reducible to A ($\forall L \in$ NP : $L \leq_p A$). In other words, a problem in this class is "at least as hard as the hardest problem in NP". To prove that a given problem B is NP-hard it suffices to show that $A \leq_p B$ for any NP-hard problem A. In addition to that, a problem A is *NP-complete* if A is NP-hard and $A \in$ NP. The NP-completeness of a problem B can be proven by showing that B is in NP and that $A \leq_p B$ for any NP-complete problem A. This is due to the fact that the $\leq_p$ relation is transitive (concatenation of the transformations). An example of NP-hard problems and their according reductions is shown in Figure 2.1.

2.1 NP-hardness of Graph Partitioning

We now show that perfectly balanced bipartitioning is NP-hard and that there is no polynomial time constant factor approximation algorithm for perfectly balanced k-partitioning with $k \geq 3$ blocks unless P=NP. The complexity of these problems serves as a justification for using approximations that may not yield optimal solutions but take a reasonable amount of time to compute.

2.1.1 Perfectly Balanced Bipartition

We present the polynomial reduction for balanced bipartitioning from [53] which reduces Simple Max Cut to the partitioning problem. This reduction is then used to prove that perfectly balanced bipartition is NP-complete. We start with the definition of the two decision problems Simple Max Cut and Balanced Bipartition.

Simple Max Cut. The Simple Max Cut problem receives a graph $G = (V, E)$ as well as a positive integer W and answers the question whether there is a cut $(S, V \backslash S)$ such that $|\{\{u, v\} \in E \mid u \in S, v \in V \backslash S\}| \geq W$.

This problem is a simplified version of the maximum cut problem (Max Cut), both of which are known to be NP-complete [53].

Balanced Bipartition. The perfectly balanced bipartition problem in decision form takes a graph $G = (V, E)$ and an integer $W > 0$ as inputs. It then answers the question whether there is a partition $V = V_1 \cup V_2$ that satisfies $V_1 \cap V_2 = \emptyset$, $|V_1| = |V_2|$ and $|\{\{u, v\} \in E \mid u \in V_1, v \in V \backslash V_1\}| \leq W$.

We now state the main result of this section:

Theorem 2 (Garey, Johnson, Stockmeyer)
Perfectly Balanced Bipartitioning is NP-complete [53].

Proof. Given a Simple Max Cut instance $G = (V, E), W > 0$, we define a Balanced Bipartition instance $G' = (V', E'), W' > 0$ as follows:

1. $V' = V \cup \{u_1, \ldots, u_{|V|}\}$
2. $E' = \{\{u, v\} \mid u, v \in V' \text{ and } \{u, v\} \notin E\}$
3. $W' = n^2 - W$.

In addition to that, the vertices $\{u_1, \ldots, u_{|V|}\}$ are defined in such a way that $V \cap \{u_1, \ldots, u_{|V|}\} = \emptyset$. This instance can be computed in polynomial time since $|V'| = 2n$ and $|E'| = n^2 - m$. We now show that this reduction is sufficient, i.e.

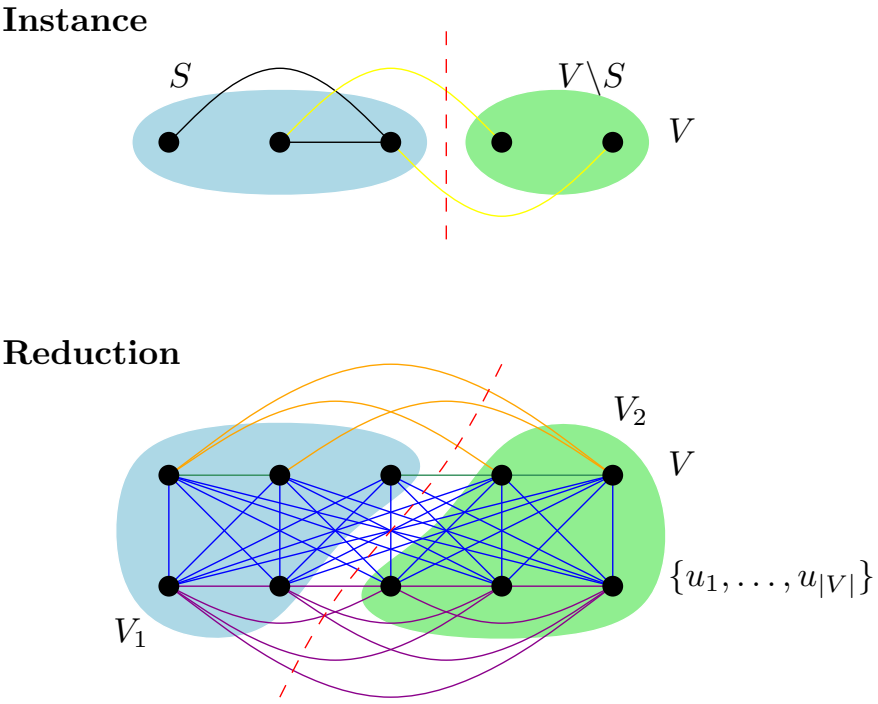


Figure 2.2: Example of a Simple Max Cut instance and the corresponding Balanced Bipartition instance. The new graph $G' = (V', E')$ contains the nodes $V' = V \cup \{u_1, \ldots, u_{|V|}\}$. The edges E' are composed of the orange, magenta and blue edges. The partitions V_1, V_2 are balanced using u_*. Thus, $|V_1| = |V_2| = n$. The resulting cut bound is $W' = n^2 - W$ with W being the number of yellow edges in the original instance.

a Simple Max Cut instance is solvable if and only if the corresponding Balanced Bipartition instance is solvable.

$\Rightarrow$: Assume there is a cut $(S, V\backslash S)$ which satisfies $|\{\{u, v\} \in E \mid u \in S, v \in V\backslash S\}| \geq W$. Since $W > 0$, we can imply that $S \neq \emptyset$ and $V\backslash S \neq \emptyset$. The idea now is to use the u_* vertices for balancing the blocks in G'. Let $j = n - |S|$ and define

1. $V_1 = S \cup \{u_1, \ldots, u_j\}$
2. $V_2 = V'\backslash V_1$.

The result is a partition $V_1 \cup V_2$ for G' with $|V_1| = |V_2| = n$. An example of this transformation can be seen in Figure 2.2.

For the cut-size induced by this partitioning we get:

$$\begin{aligned}&|\{\{u,v\} \in E' \mid u \in V_1, v \in V_2\}|\\ =&n^2 - |\{\{u,v\} \notin E' \mid u \in V_1, v \in V_2\}|\\ =&n^2 - |\{\{u,v\} \in E \mid u \in S, v \in V \backslash S\}|\\ \leq&n^2 - W = W'\end{aligned}$$

Thus, $V_1 \cup V_2$ is a solution for the Balanced Bipartition instance.

$\Leftarrow$: Assume a perfectly balanced bipartition in G' with $V' = V_1 \cup V_2, |V_1| = |V_2| = n$ such that $|\{\{u,v\} \in E \mid u \in S, v \in V \backslash S\}| \leq n^2 - W = W'$. Define the cut $S = V_1 \cap V$, $V \backslash S = V_2 \cap V$ in G by removing the u_* vertices from G'. The resulting cut-size then satisfies

$$\begin{aligned}&|\{\{u,v\} \in E \mid u \in S, v \in V \backslash S\}|\\ =&|\{\{u,v\} \notin E' \mid u \in V_1, v \in V_2\}|\\ =&n^2 - |\{\{u,v\} \in E' \mid u \in V_1, v \in V_2\}|\\ \geq&n^2 - (n^2 - W) = W\end{aligned}$$

Hence, $(S, V \backslash S)$ is a solution for Simple Max Cut.

As a final result, G has a cut with size greater or equal than W if and only if G' has a cut with size less or equal to W', that separates the vertices into to equal sized partitions. This proves our reduction and shows that perfectly balanced bipartition is NP-complete.

2.1.2 No Constant Factor Approximation for Partitioning

We now take a look at the perfectly balanced partition problem for $k \geq 3$ blocks (Balanced Partition) and show that the is no polynomial time constant factor approximation algorithm for this problem unless P=NP. For this purpose we introduce the notion of optimization problems and approximation ratios.

An *optimization problem* can either be stated as a maximization or minimization problem. We focus only on minimization problems, since one can easily transfer the following concepts to maximization problems. A minimization problem is defined via a pair $(\mathcal{L}, f)$. Here, $\mathcal{L}$ is the set of feasible solutions and $f : \mathcal{L} \to \mathbb{R}$ is the cost function. To solve a minimization problem, we look for an optimal solution $x^* \in \mathcal{L}$. A solution is optimal if and only if $f(x^*) \leq f(x)$ for all $x \in \mathcal{L}$. An algorithm for a minimization problem achieves an approximation ratio ρ if for all inputs I, it produces a solution $x(I)$ such that

$$f(x(I)) \leq \rho f(x^*(I))$$

where $x^*(I)$ denotes the optimal solution for I. The Balanced Partition problem can be stated as an optimization problem in the following way: Given a graph $G = (V, E)$ and an integer $k \geq 2$, try to find a partition $V = V_1 \cup \ldots \cup V_k$ into k disjoint subsets that satisfy $|V_1| = \ldots = |V_k| = \frac{n}{k}$. This partition is chosen in such a way that the cut-size between the blocks is minimized. The corresponding decision problem can be defined similarly to the balanced bipartition problem. We can now state the following theorem for the perfectly balanced partition problem:

Theorem 3 (Andreev and Räcke)
For $k \geq 3$ the perfectly balanced partitioning problem has no polynomial time approximation algorithm with finite approximation factor unless P = NP [5].

Proof. To prove this theorem we use a polynomial reduction of the 3-Partition problem to Balanced Partition presented in [5].
3-Partition: The 3-Partition problem receives $n = 3k$ integers $a_1, a_2, ..., a_n$ and a threshold S such that $\frac{S}{4} < a_i < \frac{S}{2}$ $(i = 1, \ldots, n)$ and

$$\sum_{i=1}^{n} a_i = kS.$$

It then answers the question if the numbers can be partitioned into triples such that each triple adds up to S. This problem is known to be *strongly* NP-complete which means that it is still NP-complete even if the numbers are bound by some polynomial in the input size n [53].

Reduction. Assume a finite factor approximation algorithm for the perfectly balanced partition problem for $k \geq 3$ and an instance of the 3-Partition problem consisting of polynomial bounded integers $a_1, \ldots, a_n$. We then construct a graph G for the balanced partition algorithm in the following way. Starting from an initially empty graph, we add a clique of size a_i $(i = 1, \ldots, n)$ for each of the integers a_i from the 3-Partition instance. An example of such a graph is shown in Figure 2.3

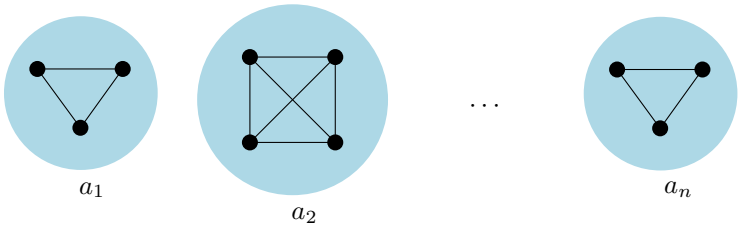


Figure 2.3: Balanced Partition instance resulting from the polynomial bounded integers $a_1, \ldots, a_n$. Here, $a_1 = a_n = 3$ and $a_2 = 4$.

Since the a_i's are polynomially bounded our graph can be constructed in polynomial time and has polynomial size. If the 3-Partition instance is solvable, we can create a k-balanced partitioning in G without cutting any edge. Otherwise, if the instance can not be solved, the optimum k-balanced partitioning has to cut at least one edge in G. Our finite factor approximation algorithm therefore only has to differentiate between these two cases. In turn, this algorithm would be able to solve the 3-Partition problem in polynomial time. This is a contradiction under the assumption that P≠NP which proves our theorem.

2.2 NP-hardness of Conductance Clustering

After examining different graph partitioning problems, we now focus on graph clustering. In particular we will look at clustering with respect to the conductance quality measure. Recall, the conductance of a clustering $\emptyset \neq S \subset V$ is defined as

$$\phi_G(S) = \frac{c_G(S)}{\min(d_G(S), d_G(V \backslash S))}$$

where $c_G(S) = |E(S, V \backslash S)|$ is the cut-size and $d_G(S) = \sum_{v \in S} d_G(v)$ the sum of the degrees in S. We prove that this problem is NP-complete by performing a polynomial reduction from the Max Cut-3 problem. Our proof is limited to the unweighted case, i.e. edges have no associated weight and is taken from [126]. For an NP-completeness proof for the weighted case the reader is referred to [125].

Conductance Clustering. The conductance clustering problem receives an undirected graph $G = (V, E)$ and a rational number $0 \leq \phi \leq 1$. It then answers the question if there is a cut $\emptyset \neq S \subset V$ such that

$$\phi_G(S) \leq \phi.$$

This problem is part of NP, since a non-deterministic algorithm can guess a cut and verify in polynomial time that $\phi_G(S) \leq \phi$. To prove that Conductance Clustering is also NP-hard, we perform a reduction from Max Cut-3.

Max Cut-3. Given a cubic graph $G = (V, E)$ and a positive integer a, Max Cut-3 answers the question if there is a cut $A \subseteq V$ such that

$$c_G(A) \geq a.$$

A cubic graph is a graph in which every vertex has a degree of three. This problem is known to be NP-complete [136].

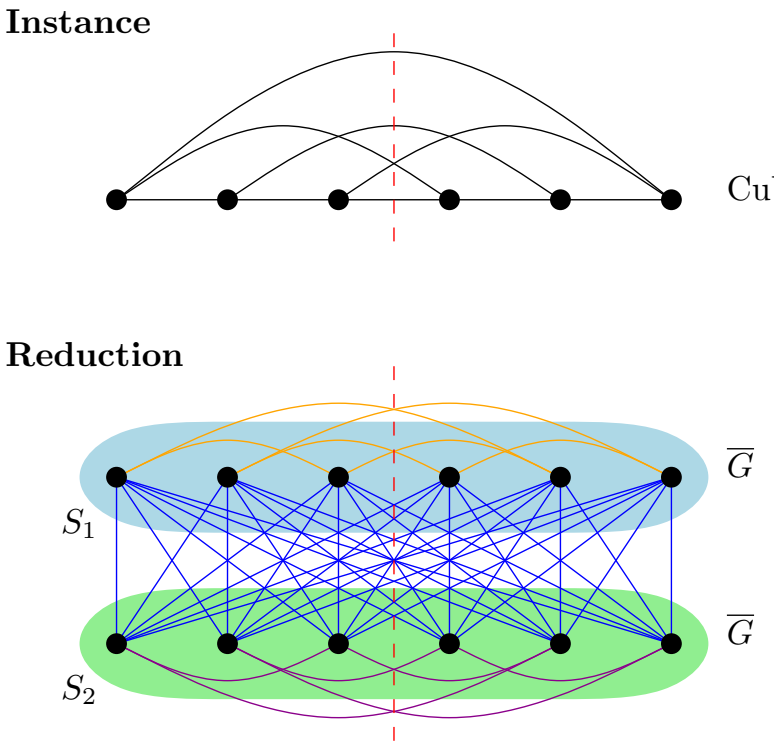


Figure 2.4: Example of a Max Cut-3 instance and its according Conductance Clustering instance. The new graph $G' = (V', E')$ is made up of two fully connected complements $\overline{G}$. The edges E' are composed of the orange, magenta and blue edges. S_1, S_2 are the projections of the cut S to V_1, V_2.

Reduction. Given a Max Cut-3 instance consisting of a cubic graph $G = (V, E)$ with $|V| = n$ and a positive integer a we construct a corresponding undirected graph $G' = (V', E')$ for Conductance Clustering. G' is composed of two fully connected copies of the complement graph of G. More specifically $V' = V_1 \cup V_2$ where $V_i = \{v^i \mid v \in V\}$ for $i = 1, 2$ and $E' = E_1 \cup E_2 \cup E_3$. $E_i = \{\{u^i, v^i\} \mid u, v \in V, u \neq v, \{u, v\} \notin E\}$ for $i = 1, 2$ are the edges in each complement of G and $E_3 = \{\{u^1, v^2\} \mid u, v \in V\}$ are the edges that connect both complements. Therefore, the number of vertices is $|V'| = 2n$ and the number of edges is $|E'| = (2n - 4)n$ since G is a cubic graph and every vertex $v \in V'$ has degree $d_G(v) = 2n - 4$. An example of such a graph can be seen in Figure 2.4. In addition to this graph, we define our new conductance bound as

$$\phi = \frac{1}{2n - 4}\left(n - \frac{2a}{n}\right).$$

It follows that the corresponding Conductance Clustering instance can be constructed in polynomial time.

Further, for a cut $\emptyset \neq S \subset V'$ with $k = |S| < 2n$, we define its projections to V_1 and V_2 in G' as

$$S_i = \{v \in V \mid v^i \in S\}$$

for $i = 1, 2$, respectively. We can safely assume that $k \leq n$ without losing generality, because $c_{G'}(S) = c_{G'}(V'\backslash S)$ and therefore $\phi_{G'}(S) = \phi_{G'}(V'\backslash S)$ based on our definition of conductance. G' is composed of two fully connected copies of the complement of G, and therefore we can follow that

$$\phi_{G'}(S) = \frac{|E'(S, V'\backslash S)|}{(2n-4) \cdot k}.$$

Since $|E'(S, V'\backslash S)| = |S| \cdot |V'\backslash S| - c_G(S_1) - c_G(S_2)$ and $|S| \cdot |V'\backslash S| = (2n-k) \cdot k$ the formula for conductance can be rewritten as

$$\phi_{G'}(S) = \frac{1}{2n-4}\left(2n - k - \frac{c_G(S_1) + c_G(S_2)}{k}\right).$$

We now prove that a Max Cut-3 instance is solvable if and only if the corresponding Conductance Clustering instance is solvable and thereby verify our reduction. Assume a Max Cut-3 instance for which a cut $A \subseteq V$ in G exists that satisfies $c_G(A) \geq a$. We then define a cut $S^A \subseteq V'$ in G' as

$$S^A = \{v^1 \in V_1 \mid v \in A\} \cup \{v^2 \in V_2 \mid V\backslash A\}$$

with its corresponding projections $S_1^A = A$ and $S_2^A = V\backslash A$. An example of this is shown in Figure 2.5. Since $|S^A| = n$ and $c_G(A) = c_G(V\backslash A)$, we can upper bound the conductance for S^A as

$$\phi_{G'}(S^A) = \frac{1}{2n-4}\left(n - \frac{2c_G(A)}{n}\right) \leq \frac{1}{2n-4}\left(n - \frac{2a}{n}\right) = \phi.$$

It follows that S^A is a solution for the Conductance Clustering instance.

Now assume a Conductance Clustering instance for which a cut $\emptyset \neq S \subset V$ in G' exists that satisfies $\phi_{G'}(S) \leq \phi$. Let $A \subseteq V$ be a maximum cut in G. We now prove that

$$\phi_{G'}(S^A) \leq \phi_{G'}(S) \leq \phi$$

for the previously defined cut S^A. We can rewrite this to

$$\frac{1}{2n-4}\left(n - \frac{2c_G(A)}{n}\right) \leq \frac{1}{2n-4}\left(2n - k - \frac{c_G(S_1) + c_G(S_2)}{k}\right)$$

where $k = |S| \leq n$. Since A is a maximum cut in G, we can assume that $2c_G(A) \geq c_G(S_1) + c_G(S_2)$ and therefore it suffices to show that

$$n - k + \left(\frac{1}{n} - \frac{1}{k}\right)(c_G(S_1) + c_G(S_2)) \geq 0.$$

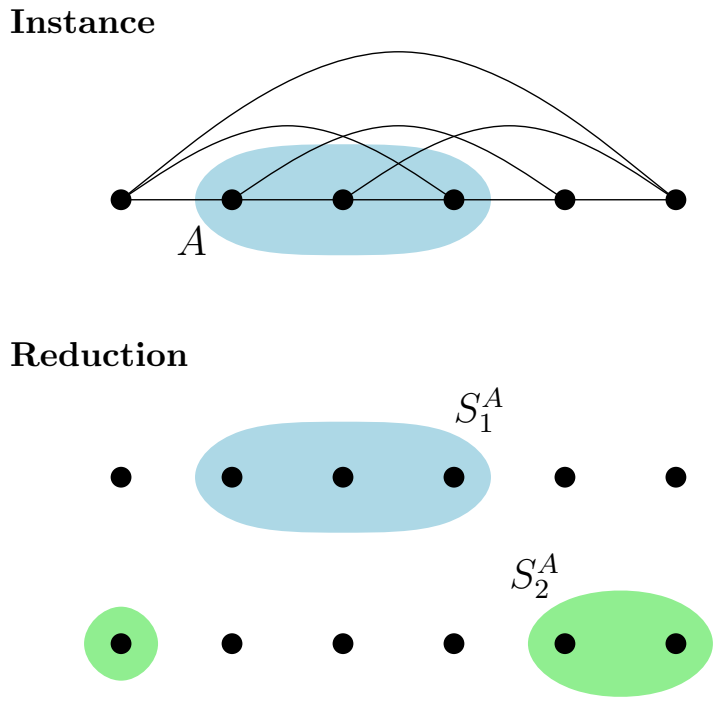


Figure 2.5: Example of a Max Cut-3 instance and the projections of its cut A.

This follows from $\frac{1}{n} - \frac{1}{k} \leq 0$ and $c_G(S_1) + c_G(S_2) \leq |S_1| \cdot n + |S_2| \cdot n = kn$. Thus,

$$\frac{1}{2n-4}\left(n - \frac{2c_G(A)}{n}\right) \leq \frac{1}{2n-4}\left(n - \frac{2a}{n}\right) = \phi.$$

Therefore, A is a solution for the Max Cut-3 instance. This verifies the correctness of our reduction and concludes our prove of the NP-completeness of Conductance Clustering.

References

This chapter is largely based on Garey et al. [53], Andreev and Räcke [5], and Šíma and Schaeffer [126]. This chapter was created by Sebastian Lamm.

Chapter 3

Integer Linear Programs

In this chapter, we explain approaches for partitioning and clustering graphs utilizing the well-known approach of (integer) linear programs. We start with an introduction into linear programs in general and then outline three specific linear programs solving the clustering and the partitioning problem.

3.1 Definitions and Basics

Definition 1 (Linear Program)
A linear program *with n variables and m constraints is defined through the following minimization / maximization problem:*

- *Linear cost function $f(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x}$, where $\mathbf{c}$ denotes a cost vector*
- *m constraints $\mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i$ such that $\bowtie_i \in \{\leq, \geq, =\}, \mathbf{a}_i \in \mathbb{R}^n$*

We get as solution space

$$\mathcal{L} = \{\mathbf{x} \in \mathbb{R}^n \mid \forall j \in 1..n : x_j \geq 0 \wedge \forall i \in 1..m : \mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i\}$$

Example: Shortest Paths. Given a graph $G = (V, E)$, it is possible to formulate a linear program whose optimum describes the lengths of all shortest paths in G starting at vertex $s \in V$. For this, we define variables d_v for each $v \in V$ denoting the total weight of the path from s to v. Consequently, we set $d_s = 0$ as a constraint. Furthermore, we want d_v to be the minimum of all neighbours plus the weight of the corresponding edge. More precisely,

$$d_v = \min_{(u,v) \in E} d_u + c(u, v)$$

We get the inequalities $d_v \leq d_u + c(u,v) \quad \forall (u,v) \in E$ and want to maximize the value of d_v. Hence, we get the following linear program:

$$\begin{aligned} \text{maximize} \quad & \sum_{v \in V} d_v \\ \text{subject to} \quad & d_s = 0 \\ & d_v \leq d_u + c(u,v) \quad \text{for all } (u,v) \in E \end{aligned}$$

Algorithms and Implementations. In 1979, Khachiyan showed that linear programs are solvable in polynomial time [81]. The worst case is

$$\mathcal{O}\left(\max(m,n)^{\frac{7}{2}}\right)$$

The implementations used in practise are much faster for most of the input graphs. However, robust and efficient implementations are very complex. In the meanwhile, several free as well as commercial software packages have been developed. Two well-known implementations of ILP solvers are the IBM CPLEX Optimizer [70] and GUROBI [58].

Integer Linear Programs and Relaxation. Besides the "standard" type of linear programs described above, there are further definitions which are in parts more restricted. However, depending on the application they are more likely to formulate a suitable program. An *integer linear program (ILP)* is a linear program with the additional constraint of all x_i being integer, formally $x_i \in \mathbb{Z}$. Often, the further restriction of a 0/1 ILP is used, meaning $x_i \in \{0,1\}$. A *mixed integer linear program (MILP)* is a linear program with some integer variables and some $x_i \in \mathbb{R}$. Besides non-restricted linear programs, solving ILPs is NP-hard. Hence, an obvious approach to solve an (M)ILP is to remove the linear constraints of a (M)ILP before solving it and map the obtained x_i to integers again. This technique is called *integer relaxation*.

3.2 Graph Clustering with ILP

We present an ILP to compute a graph clustering w.r.t. a user defined objective function. The formulation is taken from Görke [59] and is as follows:

1. We introduce decision variables $X_{uv} \in \{0, 1\}$ for each pair of vertices $u, v \in V$. X_{uv} is 0, if u and v belong to the same cluster. Formally,

$$\forall \{u, v\} \in \binom{V}{2} : X_{uv} = \begin{cases} 0 & \text{if } \mathcal{C}(u) = \mathcal{C}(v) \\ 1 & \text{otherwise} \end{cases}$$

2. We have add constraints such that our variables represent a valid clustering. More precisely, we have to ensure that the variables are reflexive, symmetric and transitive. While we get reflexivity and symmetry for free, we have to add constraints for transitivity. Transitivity means, if vertex u and vertex v are in the same cluster as well as vertex v and vertex w, u and w have to belong to the same cluster, too. We ensure this by using the following constraints:

$$\forall \{u, v, w\} \in \binom{V}{3} : \begin{cases} X_{uv} + X_{vw} - X_{uw} \geq 0 \\ X_{uv} + X_{uw} - X_{vw} \geq 0 \\ X_{uw} + X_{vw} - X_{uv} \geq 0 \end{cases}$$

3. We define a target function which has to be optimized. In this step we are free to chose a *linear* function that meets our requirements. In this case, we use modularity:

$$\mathrm{mod}_{\mathrm{ILP}}(G, \mathcal{C}_G) = \sum_{\{u,v\} \in \binom{V}{2}} \left(\mathbb{1}_{\{u,v\} \in E} - \frac{\deg(u) \cdot \deg(v)}{2 \cdot m} \right) \cdot X_{uv}$$

The proposed formulation is not the only correct one. There are countless others possible with other constraints and objectives. For instance, intra-/inter-expansion could be used as a constraint and the objective function could have multiple criteria. We refer the reader to Görke [59] for more on that. To get an idea of the runtime behavior of solving the ILP: a run with 300 vertices can take up to one day.

3.3 Balanced Partitions with ILP

We now explain ILP-based approaches for balanced graph partitioning. The first ILP tackles the bipartitioning problem and is due to Brillout [27], and the second one extends the formulation to the general k-partitioning problem (due to Christian Schulz).

3.3.1 Balanced Bipartitioning

ILP for balanced bipartitions:

1. Introduce decision variables:

$$\forall v \in V : x_v := \begin{cases} 0 & \text{if } v \text{ in block } 0 \\ 1 & \text{if } v \text{ in block } 1 \end{cases}$$

and decision variables for the edges:

$$\forall e = \{u, v\} \in E : e_{uv} := \begin{cases} 1 & \text{if } e \text{ is cut edge} \\ 0 & \text{otherwise} \end{cases}$$

2. Introduce constraints that ensure a valid partition: On the one hand, e_{uv} must only be 1, if $\{u, v\}$ is a cut edge ($x_u \neq x_v$). This can be formulated as follows:

$$\forall \{u, v\} \in E : e_{uv} \geq |x_u - x_v|$$

On the other hand, we need to ensure balance of the blocks. We do this by adding an upper and a lower bound:

$$\begin{aligned} \sum_{v \in V} x_v c(v) &\leq U \\ \sum_{v \in V} x_v c(v) &\geq L \end{aligned}$$

3. Since we want to minimize the sum of the weights of the cut edges, our objective function becomes

$$\min \sum_{\{u,v\} \in E} e_{uv} \omega(\{u, v\})$$

Note, we introduced a constraint using the $|\cdot|$ operator. This is not a linear constraint! However, we can replace it easily by

$$\begin{aligned} e_{uv} &\geq x_u - x_v \\ e_{uv} &\geq x_v - x_u \end{aligned}$$

Furthermore, we do not prevent $e_{uv} = 1$ for $x_u = x_v$ in this constraint. However, our objective function has to be minimized. Hence, if e_{uv} could be set to 0 without hitting a constraint, this will be done in order to minimize the objective function.

3.3.2 Extension to Balanced k-Partitions

As mentioned, the ILP can be extended to the arbitrary k-partitioning problem. The program becomes:

1. Introduce decision variables for the edges e_{uv}:

$$\forall e = \{u, v\} \in E : e_{uv} := \begin{cases} 1 & \text{if } e \text{ is cut edge} \\ 0 & \text{otherwise} \end{cases}$$

   and decision variables for the vertices $x_{v,k}$:

$$\forall v \in V \; \forall k : x_{v,k} := \begin{cases} 1 & \text{if } v \text{ is in block } k \\ 0 & \text{otherwise} \end{cases}$$

2. Similar constraints, adjusted with the new variables $x_{v,k}$:

$$\forall \{u, v\} \in E \; \forall k : e_{uv} \geq |x_{u,k} - x_{v,k}|$$

$$\begin{aligned} \forall k : \sum_{v \in V} x_{v,k} c(v) &\leq U \\ \forall k : \sum_{v \in V} x_{v,k} c(v) &\geq L \end{aligned}$$

   Due to the several variables $x_{v,k}$ for each vertex, we have to ensure that each vertex is assigned to exactly one block:

$$\forall v \in V : \sum_{k} x_{v,k} = 1$$

3. The objective function does not change:

$$\min \sum_{\{u,v\} \in E} e_{uv} \omega(\{u, v\})$$

As before, we have to eliminate the $|\cdot|$ operator. This can be done similar to the bipartition case. As in the clustering ILP, other objectives and more constraints etc. are possible. But unfortunately, the ILP for balanced k-partitions is too slow in practice for larger values of k.

References

This chapter is largely based on Görke [59] and Brillout [27]. The k-partitioning ILP is due to Christian Schulz. This chapter was created by Henning Schulz.

Chapter 4

Spectral Graph Partitioning

In this chapter, we have a look at spectral partitioning and clustering techniques. Spectral techniques were first used by Donath et al. [42, 43] and Fiedler [47], and have been improved subsequently by several researchers [23, 109, 127, 62, 11]. In contrast to local search approaches, it has global view on the optimization problem instead of using only local parts like the neighbourhood of a single vertex. As we will see there is a connection between eigenvectors and eigenvalues of a specific type of matrix and small cuts in a graph. More precisely, this chapter presents connections between cuts in graphs and second smallest eigenvectors/values of the matrix called Laplacian:

$$(S, V \setminus S) \overset{?}{\leftrightarrow} Lx = \lambda x$$

4.1 Mathematical Foundations

We repeat some mathematical foundations:

Definition 2 (Eigenvector and Eigenvalue)
An eigenvector *of a square matrix $A \in \mathbb{R}^{n \times n}$ is a non-zero vector $v \in \mathbb{R}^n$ that, when A is multiplied by v, yields the constant multiple of v. That is*

$$Av = \lambda v$$

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

There are families of matrices with special features concerning eigenvectors and values. One of these families are the positive semi-definite matrices:

Definition 3 (Positive Semi-Definite)
Let $A \in \mathbb{R}^{n \times n}$ be a symmetric matrix ($A^T = A$). We say A is positive semi-definite *if it verifies* ***one*** *of the following equivalent properties:*

(a) $\forall x \in \mathbb{R}^n \setminus \{\mathbf{0}\}: \quad x^T A x \geq 0$

(b) all eigenvalues of A *are positive or null.*

Proof. "(a) ⇒ (b)": Let $A \in \mathbb{R}^{n \times n}$ be a symmetric matrix such that $x^T A x \geq 0$ for all $x \in \mathbb{R}^n \setminus \{\mathbf{0}\}$. Assume, there is an eigenvector v to the eigenvalue $\lambda < 0$. Then

$$v^T A v = v^T (\lambda v) = \lambda \|v\|^2 < 0.$$

Thus, all eigenvalues have to be positive or null.

"(b) ⇒ (a)": Let $A \in \mathbb{R}^{n \times n}$ be a symmetric matrix with only non-negative eigenvalues $\lambda_1, \ldots, \lambda_k$ and corresponding eigenvectors $v_1, \ldots, v_k$. Let furthermore $x \in \mathbb{R}^n \setminus \{\mathbf{0}\}$. Then $x = \sum_{i=1}^n \alpha_i v_i$ with $\alpha_i \in \mathbb{R}$ and:

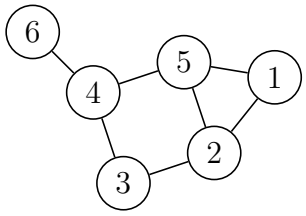
$$\begin{aligned} x^T A x &= (\sum_{i=1}^n \alpha_i v_i)^T A (\sum_{i=1}^n \alpha_i v_i) \\ &= (\sum_{i=1}^n \alpha_i v_i)^T \cdot (\sum_{i=1}^n \alpha_i A v_i) \\ &= (\sum_{i=1}^n \alpha_i v_i)^T \cdot (\sum_{i=1}^n \alpha_i \lambda_i v_i) \\ &= \sum_{i=1}^n \alpha_i^2 \lambda_i \|v_i\|^2 \geq 0 \end{aligned}$$

□

Matrices Associated with a Graph. In order to utilize the concepts described above, we have to find a matrix representations of graphs. Furthermore, these matrices should meet the requirements of the theorems in order to make use of them and the eigenvectors and values of the matrices should contain useful information. We introduce the adjacency matrix, the degree matrix and the Laplacian matrix of a graph.

Definition 4 (Adjacency Matrix)
Let $G = (V, E)$ *be a simple graph with weight function* $w : V \times V \to \mathbb{R}$. *The adjacency matrix* $A \in \mathbb{R}^{|V| \times |V|}$ *is defined as*

$$(A)_{ij} := \begin{cases} 0, & \text{if } i = j \\ w(i,j), & \text{otherwise} \end{cases}$$



$$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

Figure 4.1: Example Graph with degree, adjacency and Laplacian matrix.

In the case that G is unweighted, $w|_E \equiv 1$ and $w|_{(V\times V)\setminus E} \equiv 0$.

Definition 5 (Degree Matrix)
Let $G = (V, E)$ be a simple graph. The degree matrix $D \in \mathbb{R}^{|V|\times|V|}$ *is defined as*

$$(D)_{ij} := \begin{cases} deg(i), & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}$$

Definition 6 (Laplacian Matrix)
Let $G = (V, E)$ be a simple graph. The matrix

$$\mathcal{L} = D - A$$

is called Laplacian matrix *of G.*

Examples:

(A) In Figure 4.1, we provide the adjacency, degree and Laplacian matrix of a simple example graph.

(B) The Laplacian matrix of a *complete graph* $G = (V, E)$ is very generic: Since all vertices are connected to all others, the degree of each vertex is $n - 1$, whereat $n = |V|$. Hence, the diagonal entries of the Laplacian are $n - 1$. Furthermore, the adjacency matrix is 1 on every entry except the diagonal. Formally, the Laplacian is:

$$(L_{complete})_{ij} = \begin{cases} n - 1, & i = j \\ -1, & \text{otherwise} \end{cases}$$

(C) A *path graph* is not only a nice example but furthermore shows a connection to the Laplace operator Δ ($\Delta := -\sum \partial_{ii}^2$, here $-\frac{\partial^2}{\partial x^2}$): Consider a graph with n vertices and $(i,j) \in E \Leftrightarrow j = i+1$. The corresponding Laplacian looks like the following.

$$L_{path} = \begin{pmatrix} 1 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & 0 & -1 & 1 \end{pmatrix}$$

Let furthermore $x \in \mathbb{R}^n$ be a vector. We can formulate each entry of the product $L_{path} \cdot x$ as follows:

$$(L_{path}x)_i = -x_{i-1} + 2x_i - x_{i+1} \quad (i \neq 1, n)$$

If we consider a mapping $u : [0,1] \to \mathbb{R}$ in $C^2([0,1])$, we get the mentioned "connection" to the continuous Laplace operator. For small h, the following approximation holds:

$$\begin{aligned} u'(x) &\approx \frac{u(x+h) - u(x)}{h} \\ u'(x) &\approx \frac{u(x) - u(x-h)}{h} \end{aligned}$$

Then, the second derivative of u is approximatively for small h:

$$\begin{aligned} u''(x) &\approx \frac{u'(x+h) - u'(x)}{h} \\ &\approx \frac{u(x+h) - u(x) - (u(x) - u(x-h))}{h^2} \\ &\approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} \end{aligned}$$

Discretising $[0,1]$ with spacing h (see Figure 4.2) and setting $x_i := u(i \cdot h)$ gives us the "connection" – the Laplace operator is just a scaling on the path graph.

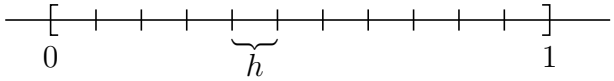


Figure 4.2: Illustration of the "spacing h".

4.2 Properties of the Laplacian Matrix

In the following we introduce and prove three theorems, which describe some important properties of a Laplacian matrix. For the sake of convenience, we only consider the unweighted case. For instance, we show that a Laplacian is symmetric if the corresponding graph is symmetric, it is positive semi-definite and has some interesting eigenvectors and eigenvalues.

Theorem 4
The Laplacian matrix of a graph G is symmetric if, and only if, G is undirected.

Proof. trivial.

Theorem 5
The Laplacian Matrix is positive semi-definite.

Proof. Let $x \in \mathbb{R}^n \setminus \{\mathbf{0}\}$.

$$
\begin{aligned}
\Rightarrow \quad (Lx)_i &= \deg(i)x_i - \sum_{(i,j)\in E} x_j \\
&= \sum_{(i,j)\in E} x_i - \sum_{(i,j)\in E} x_j \\
&= \sum_{(i,j)\in E} (x_i - x_j) \\
\Rightarrow \quad x^T Lx &= \sum_{i\in V} x_i \left(\sum_{(i,j)\in E} (x_i - x_j) \right) \\
&= \sum_{i\in V} \sum_{(i,j)\in E} (x_i - x_j) \\
&= \sum_{\{i,j\}\in E} x_i(x_i - x_j) + x_j(x_j - x_i) \\
&= \sum_{\{i,j\}\in E} \underbrace{(x_i - x_j)^2}_{\geq 0} \geq 0
\end{aligned}
$$

□

From $(Lx)_i = \sum_{(i,j)\in E}(x_i - x_j)$ follows that for each vector $x \in \mathbb{R}^n$ with $x_i = x_j$ for all $i, j \in \{1, \ldots, n\}$ holds $Lx = \mathbf{0}$. Hence, it motivates the following corollary:

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

Example: Reusing the example graph of Figure 4.1, we get

$$L_C \cdot \mathbb{1} = \begin{pmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = 0 \cdot \mathbb{1}$$

Besides the existance of the eigenvalue 0 of a Laplacian matrix, we can make further statements about it. For example, there is a connection between the multiplicity of the eigenvalue 0 and the number of connected components of the corresponding graph G, as the following theorem shows:

Theorem 6
The multiplicity of the eigenvalue 0 in the spectrum of the graphs Laplacian is the number of connected components.

Proof. "$\Rightarrow$" Let G consist of k connected components $G_1, \ldots, G_k$. Let $\mathbb{1}_i(j)$ be the characteristic vector of G_i, i.e.

$$\mathbb{1}_i(j) = \begin{cases} 1, & j \in G_i \\ 0, & \text{otherwise.} \end{cases}$$

Then: each $\mathbb{1}_i$ is an eigenvector of eigenvalue 0 and all these vectors are orthogonal to each other. (Also note $\sum_{i=1}^{k} \mathbb{1}_i = \mathbb{1}$)
"$\Leftarrow$" Let $x \in \mathbb{R}^n \setminus \{\mathbf{0}\}$ be an eigenvector to eigenvalue 0. Then:

$$x^T L x = \sum_{\{i,j\} \in E} (x_i - x_j)^2 = 0$$

Thus, the value of x must be the same across every edge and decompose the vector into k characteristic vectors (characteristic vectors of connected components). There must be at least k to span the same eigenspan. Henceforth, we assume that G is connected. $\square$

Side notes:

- L can be written as $\begin{pmatrix} L_1 & & & 0 \\ & L_2 & & \\ & & \ddots & \\ 0 & & & L_k \end{pmatrix}$
- Spectrum of L is given by the union of the spectra of the block matrices L_i
- If G is connected, then x_i needs to be constant in order for the quadratic form to vanish
- Hence, there are no more the k eigenvectors to eigenvalue 0

Lagrange Multipliers. We now outline how to tackle constraint optimization problems with Lagrangians. Given a constrained optimization problem

$$\min_{x \in \Omega} f(x) \text{ subject to } h(x) = 0$$

the Lagrangian is defined as

$$\mathcal{L}(x, \lambda) = f(x) + \lambda h(x)$$

Then solutions of the optimization problem can be obtained by using some properties of the Lagrangian: x^* is a local minimum if, and only if, there exists a unique λ^* such that

- $\nabla_x \mathcal{L}(x^*, \lambda^*) = 0$
- $\nabla_\lambda \mathcal{L}(x^*, \lambda^*) = 0$
- $y^T(\nabla^2_{xx} \mathcal{L}(x^*, \lambda^*))y \geq 0 \quad \forall y$ such that $\nabla_x h(x^*)^T y = 0$

4.3 Main Idea of Spectral Partitioning

Let $G = (V, E)$ be a graph and $(V_1, V_2 = V \setminus V_1)$ be a bisection. We define a vector $\mathbf{x} \in \{1, -1\}^n$ as

$$(x_i) := \begin{cases} 1, & i \in V_1 \\ -1, & i \in V_2 \end{cases}$$

Then, the quadratic form $\mathbf{x}^T L\mathbf{x}$ with the Laplacian matrix L of G describes the number of cut edges:

$$\mathbf{x}^T L\mathbf{x} = \sum_{\{i,j\} \in E} (x_i - x_j)^2 = 4 \cdot |E(V_1, V_2)|$$

Note, $(x_i - x_j)^2 = 0$ for $x_i = x_j$ and $(x_i - x_j)^2 = 4$ for $x_i = -x_j$. Hence, we get the following corollary:

Corollary 2
Minimizing the cut of a bisection aims at finding a vector x that minimizes $x^T Lx$.

We already know that $x = \mathbb{1}$ minimizes this equation. This corresponds to the cut $(V, \emptyset)$ (trivial cut). We need to enforce balance:

$$\begin{aligned} |V_1| = |V_2| \quad &\Leftrightarrow \quad \sum_{i=1}^{n} x_i = 0 \\ &\Leftrightarrow \quad \mathbf{x}^T \mathbb{1} = 0 \end{aligned}$$

and formulate the following optimization problem:

$$\text{(O)} \qquad \begin{aligned} \min \quad & \frac{1}{4}\mathbf{x}^T L\mathbf{x} \\ \text{subject to} \quad & \mathbf{x}^T \mathbb{1} = 0 \\ & x_i \in \{-1, 1\} \end{aligned}$$

We now relax the problem to

$$\text{(R)} \qquad \begin{aligned} \min \quad & \mathbf{x}^T L\mathbf{x} \\ \text{subject to} \quad & \mathbf{x} \in \mathbb{R}^n \\ & \|\mathbf{x}\|^2 = n \qquad \text{(instead of } \mathbf{x} \in \{-1, 1\}^n\text{)} \\ & \mathbf{x}^T \mathbb{1} = 0 \end{aligned}$$

and then rewrite it to

$$\text{(R')} \qquad \begin{aligned} \min \quad & \mathbf{x}^T L \mathbf{x} \\ \text{subject to} \quad & \mathbf{x} \in \mathbb{R}^n \\ & \|\mathbf{x}\|^2 = 1 \\ & \mathbf{x}^T \mathbb{1} = 0 \end{aligned}$$

Without loss of generality, we removed the $\frac{1}{4}$ factor of the objective in (R) and replaced $\|\mathbf{x}\|^T = n$ by $\|\mathbf{x}\|^T = 1$ in (R'). Consequently, we get different solutions for $\mathbf{x}$. However, the original $\mathbf{x}$ can be easily computed by retransforming it into the original space. We formulate the Lagrangian $\mathcal{L}$ for (R'). Since we have two constraints, we get two λ's, as well.

$$\mathcal{L}(\mathbf{x}, \lambda_1, \lambda_2) = \mathbf{x}^T L \mathbf{x} - \lambda_1(\mathbf{x}^T \mathbb{1}) - \lambda_2(\|\mathbf{x}\|^2 - 1)$$

Optimality conditions yield

$$\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \lambda_1, \lambda_2) = \quad L\mathbf{x} - \lambda_1 \mathbb{1} - \lambda_2 \mathbf{x} \quad = \mathbf{0} \tag{4.1}$$

$$\nabla_{\lambda_1} \mathcal{L}(\mathbf{x}, \lambda_1, \lambda_2) = \quad \mathbf{x}^T \mathbb{1} \quad = 0 \tag{4.2}$$

$$\nabla_{\lambda_2} \mathcal{L}(\mathbf{x}, \lambda_1, \lambda_2) = \quad \|\mathbf{x}\|^2 - 1 \quad = 0 \tag{4.3}$$

Using $\mathbf{x}^T \mathbb{1} = 0$ we derive from Equation (4.1):

$$0 = \mathbb{1}^T \nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \lambda_1, \lambda_2) = \mathbb{1}^T L \mathbf{x} - \mathbb{1}^T \lambda_1 \mathbb{1} - \mathbb{1}^T \lambda_2 \mathbf{x} = -\mathbb{1}^T \lambda_1 \mathbb{1} = -|\mathbb{1}| \lambda_1.$$

Hence we have $\lambda_1 = 0$ and therefore $L\mathbf{x} = \lambda_2 \mathbf{x}$ ((4.1)). Hence, the solution x is an eigenvector! Moreover, $\mathbf{x}^T L \mathbf{x} = \lambda_2$, since $\|\mathbf{x}\| = 1$. Thus, we can rewrite (R') to

$$\text{(R")} \qquad \begin{aligned} \min \quad & \lambda_2 \\ \text{subject to} \quad & L\mathbf{x} = \lambda_2 \mathbf{x} \\ & \mathbf{x}^T \mathbb{1} = 0 \quad (\mathbf{x} \neq \mathbb{1}) \\ & \|\mathbf{x}\| = 1 \end{aligned}$$

Since L is positive semi-definite, all eigenvalues of L are non-negative. Hence, the smallest possible eigenvalue is 0. However, this eigenvalue corresponds to the eigenvector $\mathbb{1}$ and we implicitly excluded $\mathbb{1}$ with the constraint $\mathbf{x}^T \mathbb{1} = 0$. Hence, λ_2 will be the second smallest eigenvalue of L.

We now want to derive a solution $\widetilde{\mathbf{x}}$ of (O) that is not too far away from $\mathbf{x}$, i.e. $\|\mathbf{x} - \widetilde{\mathbf{x}}\|$ should be small. To fulfill the constraint $\widetilde{\mathbf{x}}^T \mathbb{1} = 0$, $\frac{n}{2}$ of the entries of $\widetilde{\mathbf{x}}$

have to be assigned to $V_1(+1)$ and $\frac{n}{2}$ entries have to be assigned to $V_2(-1)$. The solution is to compute the median $\widetilde{m}$ of all the x_i and set the $\widetilde{x}_i$ as follows:

$$\widetilde{x}_i := \begin{cases} 1, & x_i < \widetilde{m} \\ -1, & x_i \geq \widetilde{m} \end{cases}$$

Now $\widetilde{\mathbf{x}}$ yields a partition $(V_1, V \setminus V_1)$ with

$$\forall v \in V : v \in V_1 \Leftrightarrow \widetilde{x}_v = 1$$

4.4 A Recipe for Spectral Partitioning

After we outlined the basic techniques and the main idea of spectral partitioning, we now provide a summary in the form of an algorithm. Furthermore, we discuss how to run this algorithm fast.

Basic Algorithm. The basic algorithm is shown below:

1. Build Laplace matrix L of G
2. Compute the eigenvector $\mathbf{x}$ of the smallest non-trivial eigenvalue
3. Compute the median of the x_i
4. Compute $(V_1, V \setminus V_1)$ via splitting

First of all, the Laplace matrix of G must be built. In the case of the example graph in Figure 4.3, we get

$$L = \begin{pmatrix} 2 & -1 & 0 & -1 & 0 & 0 \\ -1 & 4 & -1 & -1 & 0 & -1 \\ 0 & -1 & 4 & -1 & -1 & -1 \\ -1 & -1 & -1 & 4 & -1 & 0 \\ 0 & 0 & -1 & -1 & 2 & 0 \\ 0 & -1 & -1 & 0 & 0 & 2 \end{pmatrix}$$

Then, we have to compute the eigenvector $\mathbf{x} \neq \mathbb{1}$ of the smallest eigenvalue $\lambda_2 \neq 0$ of L. This is $\lambda_2 = 2.22$ with

$$\mathbf{x} = \begin{pmatrix} 0.28 \\ 0.19 \\ 0.08 \\ 0.11 \\ -0.90 \\ 0.24 \end{pmatrix}$$

The median of the x_i is $\widetilde{m} = 0.15$ and hence, we get get a solution for the original optimization problem (O) by setting $\widetilde{x}_i = 1 \Leftrightarrow x_i < \widetilde{m}$:

$$\widetilde{\mathbf{x}} = \begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}$$

The resulting partition $V_1, V_2 = V \setminus V_1$ can be seen in Figure 4.3.

Remarks. The second eigenvector can be computed using a modified Lanczos algorithm [87]. However, this method is expensive in terms of running time. Barnard and Simon [11] use a multilevel method to obtain a fast approximation of the Fiedler vector. The structure is similar to the multilevel method which is explained in full detail later. However, the graph is coarsened using maximal independent sets instead of edge contractions. The nodes of the maximal independent set form the nodes of the next coarser graph. On the coarsest level the Fiedler vector is computed using the Lanczos algorithm. The vector is then projected to the next finer level by setting the entries to their associated coarse nodes, if they exist, and by averaging over their neighbors in the current fine graph, otherwise. On each level the current vector is refined using a Rayleigh quotient iteration [106, 107], since it can take advantage of a good initial approximation.

Hendrickson and Leland [62] extend the spectral method to partition a graph into more than two blocks by using multiple eigenvectors which are computationally inexpensive to obtain. The method produces better partitions than recursive bisection, but is only useful for the partitioning of a graph into four or eight blocks. The authors also extended the method to graphs with node and edge weights.

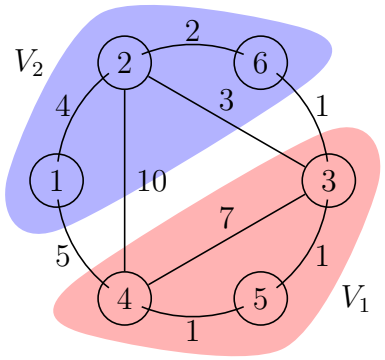


Figure 4.3: Example bisection $(V_1, V_2 = V \setminus V_1)$ of a graph $G = (V, E)$.

4.5 Spectral Modularity Maximization

Besides spectral partitioning, spectral clustering is also possible. In the following, we want to outline the basic approach. Thereby, we consider only modularity as objective and the case of only two clusters. First, we have again a look at modularity. We rewrite it as follows:

$$\begin{aligned} \mathrm{mod}(\mathcal{C}) &= \frac{1}{2m} \sum_{\substack{i,j \\ \text{in same} \\ \text{cluster}}} \left(A_{ij} - \frac{d(i)d(j)}{2m} \right) \\ &= \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \frac{d(i)d(j)}{2m} \right) \delta(C(i), C(j)) \end{aligned}$$

$\delta(C(i), C(j)) = 1 \Leftrightarrow i = j$ and $\delta(i, j) = 0$ otherwise and $C(i) :=$ the cluster of vertex i.
Then, define a vector $\mathbf{s} \in \mathbb{R}^n$ with

$$s_i := \begin{cases} 1 & \text{vertex } i \text{ is in cluster } 1 \\ -1 & \text{vertex } i \text{ is in cluster } 2 \end{cases}$$

With this definition, we can rewrite the $\delta(i, j)$ as

$$\delta(i, j) = \frac{s_i s_j + 1}{2}$$

and insert it into the rewritten modularity:

$$\begin{aligned} \mathrm{mod}(\mathcal{C}) &= \frac{1}{4m} \sum_{i,j} \left(A_{ij} - \frac{d(i)d(j)}{2m} \right) (s_i s_j + 1) \\ &= \frac{1}{4m} \sum_{i,j} \left(A_{ij} - \frac{d(i)d(j)}{2m} \right) s_i s_j \\ &= \frac{1}{4m} \sum_{i,j} (B_{ij} s_i s_j) = \frac{1}{4m} \mathbf{s}^T B \mathbf{s} \end{aligned}$$

with $B_{ij} := A_{ij} - \frac{d(i)d(j)}{2m}$. Afterwards, we relax the integer constraint for $\mathbf{s}$ with real numbers and $\mathbf{s}^T\mathbf{s} = n$. The Lagrangian of this optimization problem is

$$\mathcal{L}(s, \lambda) = \mathbf{s}^T B \mathbf{s} + \lambda(n - \mathbf{s}^T \mathbf{s})$$

The optimality conditions yield:

$$B\mathbf{s} = \lambda \mathbf{s}$$

If we insert this equation into the rewritten modularity equation, we get

$$\mathrm{mod}(\mathcal{C}) = \frac{1}{4m}\lambda \mathbf{s}^T \mathbf{s}$$

Hence, λ will correspond to the largest eigenvalue, since we want to maximize modularity. In practice, a clustering $\widetilde{\mathbf{s}}$ is chosen close to the result vector $\mathbf{s}$, for example by

$$\widetilde{s_i} := \begin{cases} 1 & s_i > 0 \\ -1 & \text{otherwise} \end{cases}$$

References

This chapter is largely based on [11, 68, 100]. This chapter was created by Henning Schulz.

Chapter 5

Multilevel Graph Partitioning

The basic idea of multilevel graph partitioning can be traced back to multigrid solvers for solving systems of linear equations [130], but more recent practical methods are based on mostly graph theoretic aspects, in particular edge contraction and local search. As already mentioned, the method was initially introduced to the graph partitioning area by Barnard and Simon [11] to speed up spectral partitioning techniques. Hendrickson and Leland [61] formulated the multilevel approach as it is known today. Already a decade earlier, Bui et al. [28] remarked that a two level approach, i.e. randomly contracting edges, improves the result of a partitioning algorithm if it is applied on the coarse graph.

Before we outline the multilevel approach, we need to define the notion of edge contractions. *Contracting* an edge $\{u, v\}$ means to replace the nodes u and v by a new node x connected to the former neighbors of u and v. We set $c(x) = c(u)+c(v)$ so that the weight of a node at each level is the number of nodes it is representing in the original graph. If replacing edges of the form $\{u, w\}$, $\{v, w\}$ would generate

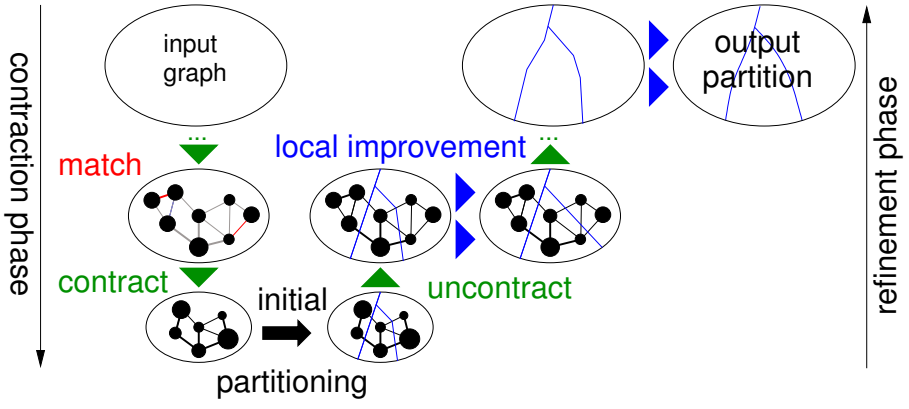


Figure 5.1: The multilevel approach to graph partitioning. Source: [117].

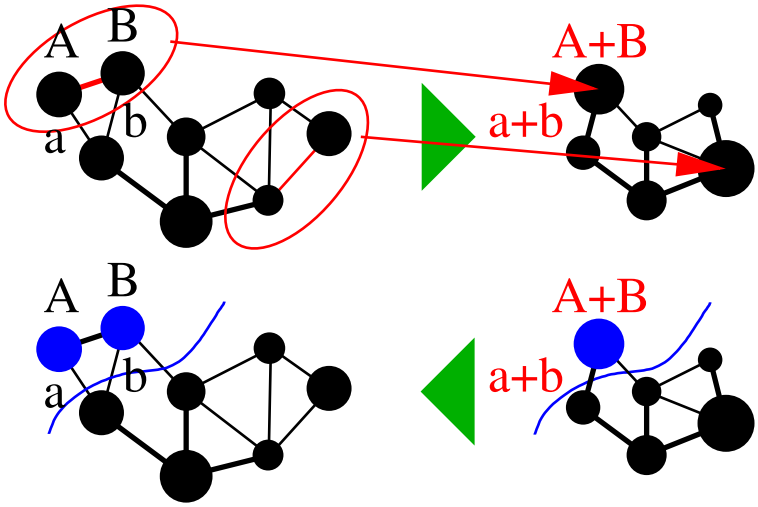


Figure 5.2: A example matching highlighted in red and contraction of matched edges. After a partition has been computed on the coarser graph, it can be transferred to the finer level by putting the finer nodes into the partition of their coarse representative. Due to the way the contraction is defined, edge cut and balance are the same after solution transfer. Source: [122].

two parallel edges $\{x, w\}$, a single edge with $\omega(\{x, w\}) = \omega(\{u, w\}) + \omega(\{v, w\})$ is inserted. *Uncontracting* an edge e undoes its contraction. In order to avoid tedious notation, G will denote the current state of the graph before and after a (un)contraction unless we explicitly want to refer to different states of the graph.

The *multilevel approach* to graph partitioning consists of three main phases. It is outlined in Figure 5.1. In the *contraction* (coarsening) phase, a hierarchy of graphs is created. There are multiple ways to do that. The most common way is to iteratively identify matchings $M \subseteq E$ and contract the edges in M. Contraction should quickly reduce the size of the input and each computed level should reflect the global structure of the input network. An example matching that is contracted is shown in Figure 5.2. Contraction is stopped when the graph is sufficiently small to be directly partitioned using some expensive other algorithm which were described in the previous sections such as spectral partitioning, graph growing or bubbling. In the *local improvement* (or uncoarsening) phase, the matchings are iteratively uncontracted. Note that due to the way that the contraction is defined, a partitioning of the coarse level creates a partitioning of the finer graph having the same objective and balance. After uncontracting a matching, a local improvement algorithm moves nodes between blocks in order to improve the cut size or balance. Usually variants of the Fiduccia-Mattheyses algorithm are used. The intuition behind this approach is that a good partition at one level will also be a good partition on the next finer

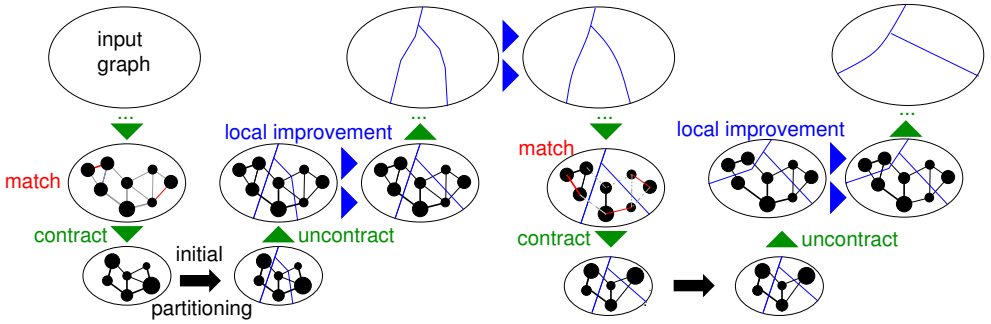


Figure 5.3: An illustration of iterated V-cycles. In this case, the multilevel scheme is iterated two times. During the coarsening of the second iteration, cut edges are not contracted and the input partition is used as initial partition of the coarsest graph. Source: [117].

level, so that local search will quickly find a good solution. Moving a node on a coarse level hierarchy usually corresponds to the movement of a whole set of node movements of the finest level of the hierarchy. Intuitively, the multilevel scheme has a global view on the optimization problem on the coarse levels of the hierarchy and a very local view on the finest levels with respect to the original graph.

It is worth mentioning that there are recursive partitioning approaches that use the multilevel approach for the bisection of a graph by Karypis et al. [77], and that they were the first who had a linear time $\mathrm{O}(m)$ implementation of this scheme to obtain a k-partition [78] (using recursive multilevel bisection only on the coarsest level and a direct k-way local search algorithm).

In the chapters that follow, we will go into detail of each component of the multilevel scheme. We will will learn about different local search algorithms, matching algorithms and other algorithms to perform contraction as well as initial partitioning algorithms and meta-heuristics.

Iterated Multilevel Algorithms. A common approach to obtain high quality partitions is to use a multilevel algorithm multiple times using different random seeds for initialization of the coarsening and local search algorithms. One can then simply use the best partition that has been found.

An improvement of this method are Iterated Multilevel Algorithms (V-cycles) which were introduced by Walshaw [134]. The main idea is to iterate coarsening and local search several times, again using different seeds for random tie breaking. However, instead of performing a full restart, one can use the information/partition that has already been obtained. More precisely, after the first iteration of the multilevel scheme is done, one performs additional iterations such that cut edges are not contracted. Thus a given partition can be used as initial partition of the coarsest

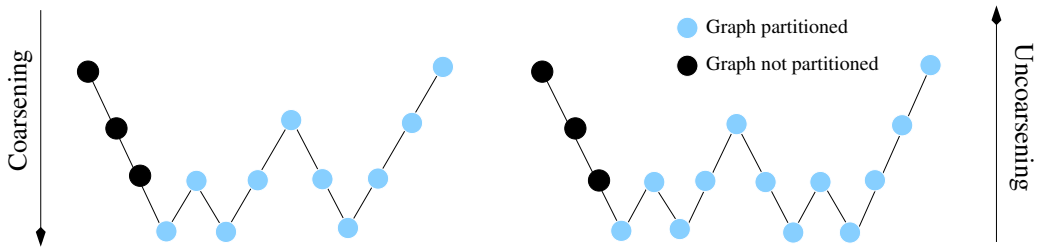


Figure 5.4: An abstract view of F-cycle (left) and W-cycle (right). Edges that run between blocks are not contracted as soon as the graph is partitioned. Source: [117].

graph (having the same balance and cut as the partition of the finest graph). This ensures non-decreasing partition quality, if the local search algorithm guarantees no worsening. The concept of iterated V-cycles is illustrated in Figure 5.3. Note that local search algorithms have, to some extent, the ability to climb out of local minima and due to the randomization of the coarsening, the hierarchies created in later cycles are different. That introduces further diversification for local search.

In multigrid linear solvers, Full-Multigrid methods are preferable to simple V-cycles [26]. Therefore, one can also use the following two global search strategies namely *W-cycles* and *F-cycles* for graph partitioning. A W-cycle works as follows: on *each* level we perform *two recursive calls* using different random seeds for contraction and local search. As soon as the graph is partitioned, edges that are between blocks are not contracted. An F-cycle works similar to a W-cycle with the difference that further recursive calls are only made the second time that the algorithm reaches a particular level. Figure 5.4 gives an abstract view.

In most cases, the partitions of the coarsest graph obtained in a second iteration of the V-cycle by an initial partitioner are much worse than the partition that is used on the coarsest graph (the partition of the finest graph that is given). In other words, in a second iteration of a V-cycle the initial partitioner is not able find better partitions of the coarsest graph than the one that is applied. Therefore no further initial partitioning is used as soon as the graph is partitioned. Experiments in [117] show that all cycle variants are more efficient than simple restarts of the algorithm.

In order to bound the execution time of the more sophisticated schemes, we a level split parameter d is used such that further recursive calls are only performed every dth level.

Analysis. We now roughly analyze the running time of different global search strategies under a few assumptions. The shrink factor a names the factor that the graph shrinks (nodes and edges uniformly) during one coarsening step.

Theorem 7
If the time for coarsening and uncoarsening is $T_{cr}(n) := bn$ and a constant shrink factor $a \in [1/2, 1)$ is given, then:

$$T_W(n) \begin{cases} \leq \frac{1}{1-2a^d} T_V(n) & \text{if } 2a^d < 1 \\ \in \Theta(n \log n) & \text{if } 2a^d = 1 \\ \in \Theta(n^{\log_{d \log 1/a} 2}) & \text{if } 2a^d > 1 \end{cases}$$

$$T_F(n) \leq \frac{1}{1-a^d} T_V(n),$$

where T_V is the time for a single V-cycle and T_W,T_F are the time for a W-cycle and F-cycle with level split parameter d.

Proof. The running time of a single V-cycle is given by

$$T_V^\ell(n) = \sum_{i=0}^{\ell} T_{\text{cr}}(a^i n) = bn \sum_{i=0}^{\ell} a^i = bn \frac{1-a^{\ell+1}}{1-a},$$

where ℓ is the number of levels of the cycle. The running time of a W-cycle with level split parameter d is given by the time for d coarsening and uncoarsening steps plus the time for the two recursive calls on the coarsest of the just created graphs. For the case $2a^d < 1$, we get

$$\begin{aligned} T_W^\ell(n) &= bn \sum_{i=0}^{d-1} a^i + 2T_W(a^d n) = bn \sum_{k=0}^{\ell/d} 2^k \sum_{i=0}^{d-1} a^{k \cdot d} a^i \\ &= bn \sum_{i=0}^{d-1} a^i \sum_{k=0}^{\ell/d} (2a^d)^k \leq T_V^\ell(n) \sum_{k=0}^{\infty} (2a^d)^k \\ &= \frac{1}{1-2a^d} T_V^\ell(n). \end{aligned}$$

The other two cases for the W-cycle follow directly from the master theorem for analyzing divide-and-conquer recurrences. To analyze the running time of an F-cycle, we observe that

$$\begin{aligned} T_F^\ell(n) &\leq \sum_{i=0}^{\ell} T_{\text{V}}^i(a^{i \cdot d} n) = \sum_{i=0}^{\ell} b a^{i \cdot d} n \frac{1-a^{i+1}}{1-a} = \frac{bn}{1-a} \sum_{i=0}^{\ell} a^{i \cdot d}(1-a^{i+1}) \\ &\leq bn \frac{1-a^{\ell+1}}{1-a} \sum_{i=0}^{\infty} (a^d)^i = \frac{1}{1-a^d} T_V^\ell(n). \end{aligned}$$

This completes the proof of the theorem.

Chapter 6

Local Search

In general, given a partition of a graph, a *local search algorithm* aims to improve an objective function (such as the number of edges that run between blocks) by moving nodes between the blocks. With the application of improving the paging properties of computer programs in mind, Kernighan and Lin [79] were probably the first that defined the graph partitioning problem and worked on local improvement methods for this problem. In his PhD thesis [80] from 1969, Kernighan states *"A program [...] can be thought of as [...] a set of connected entities. The entities might be subroutines, [...]. The connections between the entities might represent [...] references by one entity to another. The problem is to assign the objects to "pages" (of a given size) to minimize the number of references between objects which lie on different pages."* and defines methods to find and improve a partition of a graph. The main idea of Kernighan and Lin was that, given a balanced partition of a graph into two blocks V_1 and V_2, there are subsets $A \subset V_1$ and $B \subset V_2$ such that the partition that is created when moving the nodes in A to V_2 and the nodes in B to V_1, is globally optimal. Kernighan and Lin then contributed a method to find "good" sets A and B to reduce the cost of a partition. One major drawback of the method is that it is expensive in terms of asymptotic running time. The worst case running time for an iteration of the Kernighan-Lin algorithm is $\mathrm{O}(n^2 \log n)$. Dutt [45] has shown how to improve this to $\mathrm{O}(m \max(\log n, \Delta))$ time. A major breakthrough has been achieved by Fiduccia and Mattheyses [46] in 1982. Fiduccia and Mattheyses modified the algorithm and data structures such that the asymptotic running time of their local search algorithm was reduced to linear time $\mathrm{O}(m)$.

The rest of the chapter is organized as follows. After presenting general local search, we explain details of the KL and FM algorithm. We continue with commonly used optimizations as well as generalizations. We then put our attention on more sophisticated local search schemes that introduce a large amount of localization, build models to enlarge the neighborhood of local search and get to know algorithms that are based on the computation of maximum flows.

6.1 Introduction to Local Search

Local search algorithms aim to improve a given solution of an optimization problem. More precisely, given a space $\mathcal{L}$ of solutions and an initial solution $x \in \mathcal{L}$, a local search algorithm explores solutions in the neighborhood of the current solution that are within the solution space, trying to find a better solution. It is often a question of defining the right neighborhood to make a local search algorithm good and efficient. On the one hand, one has to be able to efficiently explore the neighborhood of a given solution to obtain a fast local search algorithm. On the other hand, the neighborhood has to be large enough so that the algorithm does not get stuck in local minima to quickly. Later in this chapter, we will see an example in which increasing expanding the neighborhood of solution improves solution quality of the local search algorithm. If the neighborhood is not chosen properly this can prevent the algorithm from finding a good solution. Figure 6.1 and Figure 6.2 show an examples of neighborhoods where it is impossible to reach the global optimum by a local search if the starting point is not chosen right.

A commonly used variation of local search is hill climbing which looks for a better solution in the neighborhood of the current best solution until no better solution can be found in the current neighborhood.

Algorithm 1 Local Search

find some feasible solution $x \in \mathcal{L}$
$\hat{x} := x$
while not satisfied with $\hat{x}$ **do**
  $x :=$ some heuristically chosen element from $\mathcal{N}(x) \cap \mathcal{L}$
  if $f(x) < f(\hat{x})$ **then**
    $\hat{x} := x$
  end if
end while

Algorithm 2 Hill Climbing

find some feasible solution $x \in \mathcal{L}$
$\hat{x} := x$
Loop
if $\exists x \in \mathcal{N}(x) \cap \mathcal{L} : f(x) < f(\hat{x})$ **then**
  $\hat{x} := x$
else
  return $\hat{x}$
end if

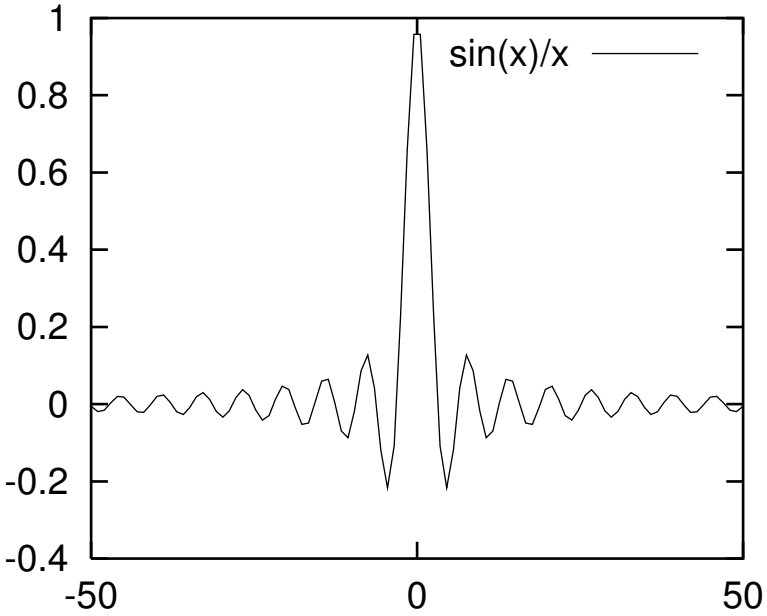


Figure 6.1: There can be a large amount of local optima while the global optimum can be orders of magnitude better than most local optima

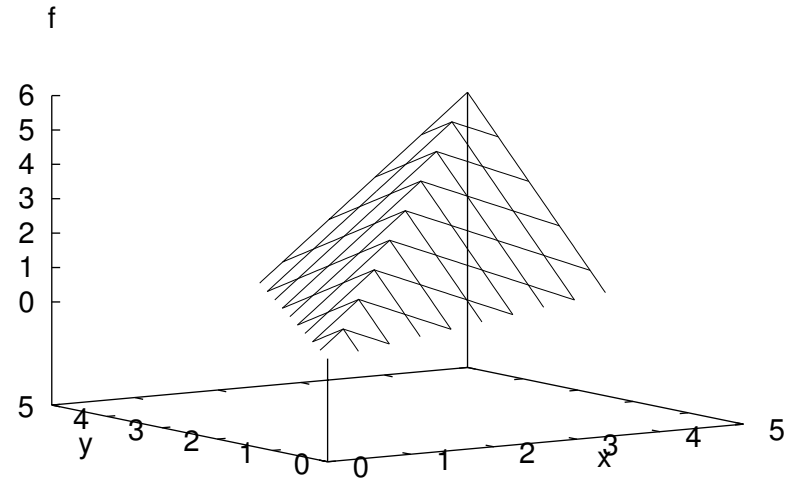


Figure 6.2: Example of a neighborhood that makes it hard to find the global optimum by using local search

6.2 Kernighan-Lin Algorithm

As mentioned earlier, the Kernighan-Lin algorithm [79] tries to improve a given partition by finding subsets $A \subset V_1$, $B \subset V_2$ and then moving the nodes in A and B to the respective opposite block. Indeed, there are optimal choices for A and B, but finding them is also NP-hard [79] (why?). Hence, Kernighan and Lin developed a heuristic approach to find "good" sets. One *pass* of the Kernighan-Lin algorithm consists of finding these sets and exchanging the corresponding nodes. The Kernighan-Lin algorithm repeatedly finds such sets A, B to be exchanged until it reaches a local optimum, i.e. exchanging the sets does not decrease the number of edges cut.

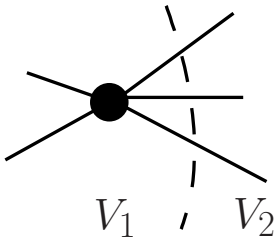


Figure 6.3: A node with gain one.

We now explain how Kernighan and Lin perform a single pass. Before we start, we introduce the definition of a node's gain. Let the input graph G be partitioned into two blocks V_1 and V_2. The *gain* of a node $v \in V_1$ is defined as $g(v) = \omega(\{(v,w) \mid w \in \Gamma(v) \cap V_2\}) - \omega(\{(v,w) \mid w \in \Gamma(v) \cap V_1)$, i.e. the reduction in the cut when v is moved from block V_1 to block V_2. The notion of gain is used analogously for nodes in block V_2. Thus, when $g(v) > 0$ we can decrease the cut by $g(v)$ by moving v to the opposite block. Figure 6.3 gives an example. Since the partition should stay balanced, the Kernighan-Lin algorithm searches for pairs of nodes to be exchanged. For $v \in V_1$ and $w \in V_2$ let $g(v,w)$ denote the gain of exchanging v and w between V_1 and V_2. Analogously, if v and w are not adjacent, then the gain is $g(v,w) = g(v) + g(w)$. If v and w are adjacent, then $g(v,w) = g(v) + g(w) - 2\omega(v,w)$ since the edge between v and w will still be a cut edge after the nodes are exchanged. Now a pass works as follows. First of all, a node can have two states: marked and unmarked. At the beginning of a pass each node is unmarked. Then, the following procedure is repeated p times where $p = \min(|V_1|, |V_2|)$. Find an unmarked pair $v \in V_1$ and $w \in V_2$ for which $g(v,w)$ is maximum. Note that $g(v,w)$ is not necessarily positive. Mark v and w and update the gain values of all the remaining unmarked nodes as if we had exchanged v and w. Only the gain values of the neighbors of v and w must be updated, since only these values could have changed. The step of finding a pair such that $g(v,w)$ is maximum can be implemented in $\mathrm{O}(n \log n)$ time.

After this procedure is done, we have an ordered list L of node pairs (v_i, w_i), $i = 1, \ldots, p$. To output the sets A and B, we first find the smallest index $k \in \{0, \ldots, p\}$ such that $\sum_{i=1}^{k} g(v_i, w_i)$ is maximum. The sets are then defined as $A := \cup_{i=1}^{k} \{v_i\}$ and $B := \cup_{i=1}^{k} \{w_i\}$. If k is not zero, then the cut will be reduced if A and B are exchanged. In this case, the exchange is done and a new pass is started. Note that the algorithm has the ability to climb out of local minima to a

Algorithm 3 Kernighan-Lin Local Search

Data: $G = (V, E)$, initial bisection $\{V_1, V_2\}$
for all $v \in V$ **do**
  compute $g(v)$
end for
repeat
  ordered list $L \leftarrow \emptyset$
  unmark all vertices $v \in V$
  for $i = 1$ **to** $n = \min(|V_1|, |V_2|)$ **do**
    $(v_1, v_2) \leftarrow \text{argmax}_{unmarked v_1 \in V_1, v_2 \in V_2} g(v_1, v_2)$
    update g-values for all $v \in N(v_1) \cup N(v_2)$
    append (v_1, v_2) to L and mark v_1, v_2
  end for
  $j \leftarrow \text{argmax}_k \Phi(k)$
  $\gamma \leftarrow \Phi(j)$
  if $\gamma > 0$ **then**
    exchange the first j vertex pairs
  end if
until $\gamma \leq 0$

certain extent due to the way in which the sets A and B are created. This is one of the key features of the algorithm.

6.3 Fiduccia and Mattheyses

Over time there have been many improvements made to the Kernighan-Lin algorithm. The most important improvement is a slight modification of the algorithm and the reduction in running time that was provided by Fiduccia and Mattheyses [46] in 1982. They reduce the complexity for a single pass to $O(m)$ by using novel data structures. Like the Kernighan-Lin method, the Fiduccia-Mattheyses method performs passes in which each node is moved at most once, and the best bisection observed during an iteration (if the corresponding reduction in the number of edges cut is positive) is used as input for the next iteration. However, instead of selecting pairs of nodes, the Fiduccia-Mattheyses method selects single nodes for movement.

In the perfectly balanced case, a pass of the Fiduccia-Mattheyses method works as follows. The algorithm starts by setting the state of every node to unmarked. In each step an unmarked node v with maximum gain value is *alternately* selected from the blocks V_1 and V_2. The node is then marked and the gain values of its unmarked neighbors are updated. This leads to two ordered sequences $(v_1, \ldots, v_p)$

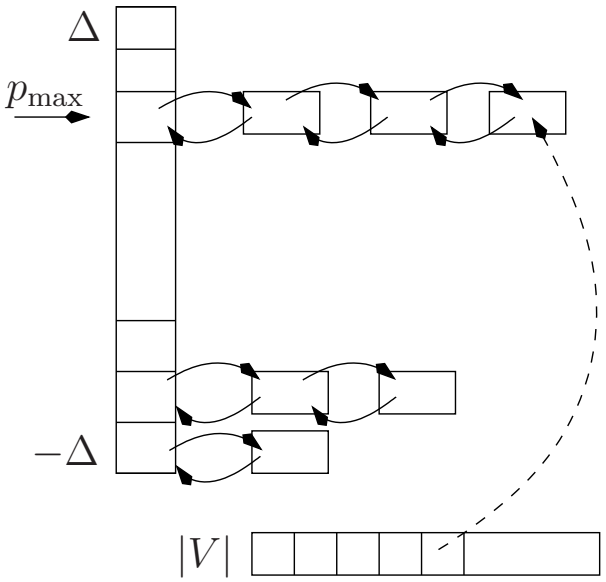


Figure 6.4: Bucket priority queue used in the Fiduccia-Mattheyses algorithm.

and $(w_1, \ldots, w_p)$ with $v_i \in V_1$ and $w_i \in V_2$. The algorithm then searches for the smallest index $k \in \{0, \ldots, p\}$ such that $\sum_{i=1}^{k} g(v_i) + g(w_i)$ is maximized. If the resulting sum is positive, the algorithm performs the movements and starts the next pass. Otherwise, the algorithm stops. If a reasonable amount of imbalance is allowed, then instead of alternately selecting the blocks, the balance criterion is used for the selection of the block. Note that both, the Kernighan-Lin and the Fiduccia-Mattheyses algorithm, do not stop when the corresponding sums are negative and thus are able to climb out of local minima to a certain extent.

In comparison to the Kernighan-Lin algorithm, there are two major modifications. First, nodes are selected independently for exchange so that computation of the gain values can be done efficiently. On the other hand, Fiduccia and Mattheyses provide a data structure such that the node with the best gain and the update of the gain values of the neighbors of a moved node can be done in constant time if the edge weights are non-negative integers. The data structure that is used to achieve this is a bucket priority queue. We now briefly outline this data structure since we also use it in our local search algorithms. Figure 6.4 gives an illustration. Let us assume that the graph has unit edge weights (integer weights are introduced straightforwardly). First, observe that the largest gain that a node can have is smaller than or equal to the maximum degree of a node (a similar argument holds for the smallest gain value that a node can have). Hence, one needs at most $2\Delta + 1$ buckets to order and to maintain all nodes sorted by their gain. Now, one needs two such specialized bucket queues, one for each block. Let

the buckets be numbered/ordered in the following way: $[-\Delta, \ldots, \Delta]$. In the ith bucket a doubly linked list[1] stores all nodes that have gain i. Furthermore, for each node the position within the linked list of its bucket and a pointer $p_{\max}$ to the largest/maximum non-empty bucket is stored separately. Clearly, removing a node from and inserting a node into its bucket can be done in constant time. Hence, if the gain of a node changes, then the position of the node in the bucket queue is also updated in constant time. The pointer to the maximum element is updated in constant time if a node is inserted into the queue and the gain is larger than the current maximum. When trying to remove a node v with maximum gain, one might have to update the pointer to the maximum non-empty bucket by decreasing the pointer until one has found the largest non-empty bucket since the pointer is not updated when a node is removed. This can also be done in amortized constant time. To see this, let v and w be two subsequent max gain nodes selected for movement by the Fiduccia-Mattheyses algorithm that are initially in the same block. The cost for updating the pointer $p_{\max}$ can be at most $\mathrm{O}(d(v) + d(w))$ (this is the case when $g(v) = d(v)$ and $g(w) = -d(w)$). However, this is amortized, since the gain values of all neighbors of both maximum elements have to be updated. Hence, the total running time of one pass of the Fiduccia-Mattheyses algorithm is in $\mathrm{O}(m)$. It is worth mentioning that Bob Darrow was the first who implemented the Fiduccia-Mattheyses algorithm (he is mentioned only in the acknowledgements of the paper of Fiduccia and Mattheyses [46]). An example execution of the algorithm can be found in Figure 6.5.

On small randomly generated graphs, the quality of the partitions produced by the Fiduccia-Mattheyses algorithm is slightly worse than the quality produced by the Kernighan-Lin algorithm [72]. The graphs used had about five hundred nodes and a random initial partition of the graph was used as starting point of the algorithms.

[1] usually arrays are used in real implementations

Algorithm 4 Fiduccia-Mattheyses Local Search

Data: $G = (V, E)$, initial bisection $\{V_1, V_2\}$
for all $v \in V$ **do**
  compute $g(v)$
end for
repeat
  for all $v \in V_1$ **do**
    put $g(v)$ into bucket queue $\mathcal{Q}_1$
  end for
  for all $v \in V_2$ **do**
    put $g(v)$ into bucket queue $\mathcal{Q}_2$
  end for
  ordered list $L \leftarrow \emptyset$
  unmark all vertices $v \in V$
  while $\exists$ unmarked vertex **do**
    alternatively select queue $\mathcal{Q} \in \{\mathcal{Q}_1, \mathcal{Q}_2\}$
    $v \leftarrow$ maxgain vertex in $\mathcal{Q}$
    move v to other block, append it to L, mark v
    remove v from $\mathcal{Q}$
    for all $w \in N(v)$ **do**
      update $g(w)$
    end for
  end while
  $j \leftarrow \text{argmax}_l \tilde{\Phi}(l) := \sum_i^l g(v_i) + g(w_i)$
  $\gamma \leftarrow \tilde{\Phi}(l)$
  if $\gamma > 0$ **then**
    apply changes
  end if
until $\gamma \leq 0$

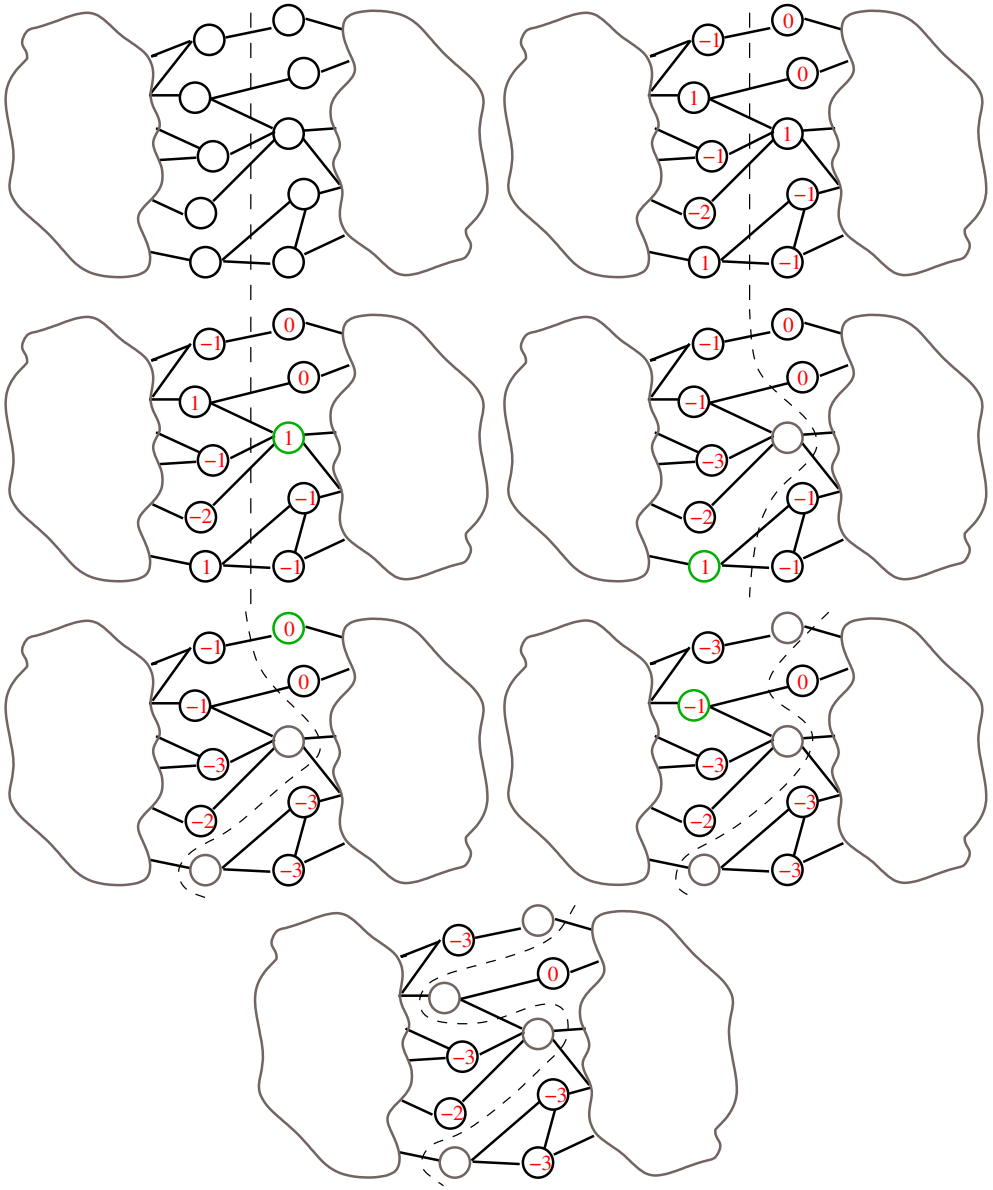


Figure 6.5: Example of the FM algorithm. The algorithm starts with a given bisection (1) and computes the gain values for all vertices (2) (here only boundary vertices are illustrated). Next the vertex with the highest gain value from the right block is selected (3) and moved to the other block while updating the gain values of all its neighbors (4). Afterwards the vertex with the highest gain value from the left block is selected (4) and moved to to right block, updating the gain values of all neighbors (5). In pictures 5 to 7 these steps are repeated.

Further Improvements

Karypis and Kumar [78] successfully improved the algorithm further by introducing the following modifications. First of all, only boundary nodes are kept in the priority queues and the moves are done right away. After local search has stopped, they undo the node movements until they are at the best partition found during the iteration. On the other hand, their variant of the algorithm terminates when the edge cut does not decrease after x node moves. Terminating the Fiduccia-Mattheyses pass in this fashion significantly reduces the running time. However, note that *all* boundary nodes are used for initialization.

The third modification that has been done by Karypis and Kumar is the following. First note that the Fiduccia-Mattheyses algorithm is dependent on the order in which the gain values are inserted into the priority queue, and a random tie-breaking mechanism is used if there are multiple nodes with equal gain. Hence, an additional pass of the algorithm can yield an improved cut even if a previous pass did not yield an improvement. Karypis and Kumar used multiple restarts of the algorithm to improve solution quality.

The Fiduccia-Mattheyses algorithm leaves the algorithm designer with the freedom to choose the block from which a node shall be moved to the opposite block. Indeed, one has to take the balance constraint into account. However, if the imbalance parameter is not too small, there are a number of possibilities. In Sanders et al. [66], we implemented three different block selection strategies which improved the quality of the output partition. The first strategy selects nodes with maximum gain from the blocks alternately, the second strategy always selects the block with maximum size, and the third strategy always uses the block where the node has the larger gain value (with respect to the balance constraint). Note that the first strategy does not alter the balance of a partition, the second strategy may improves the balance and the third strategy may worsen the balance of the input partition but stays within the balance constraint.

Helpful Sets by Diekmann et al. [40, 99], introduce a more sophisticated neighborhood relation for the exchange of nodes in the bipartitioning case. These algorithms are inspired by a proof technique of Hromkovič and Monien [69] for proving upper bounds on the bisection width of a graph. Instead of migrating single nodes, whole sets of nodes are exchanged between the blocks to improve the cut. Selecting such node sets is based on the notion of helpfulness. The helpfulness of a node set is the reduction in the cut if the node set is moved to the opposite block. More precisely, a node set is called ℓ-helpful if it reduces the cut by ℓ. One round of the algorithm works as follows. First, the algorithms tries to find an ℓ-helpful set in *one* block of the partition. Then the algorithm tries to find a balancing set in the opposite block. A balancing set has the same cardinality as the found ℓ-helpful set and is at least $-\ell + 1$ helpful. If such sets can be found,

the movements are performed, and the algorithm proceeds with the next round. Otherwise, the algorithm stops. The running time of the algorithm is comparable to the Kernighan-Lin algorithm while solution quality is often better than other methods [99].

Extension to k-way Local Search

A common method to create a k-partition is recursive bisection [80]. The graph is recursively divided into two blocks (this includes two-way local search) until the number of blocks is reached, i.e. a bisection algorithm is used to split the graph into two blocks. The same is done on the graphs induced by the two created blocks until the desired number of blocks is reached. It has been shown by Simon and Teng [128] that, due to the lack of global knowledge, recursive bisection can create partitions that are very far away from the optimal partition so that there is a need for k-way local search algorithms.

There are multiple ways of extending the Fiduccia-Mattheyses algorithm to get a local search algorithm that can improve a k-partition. Given a k-partition of a graph, the first obvious idea is to use a two-way local search algorithm between all pairs of blocks that share a non-empty boundary, i.e. blocks that are connected by at least one edge. One early extension of the Fiduccia-Mattheyses algorithm to k-way local search was described by Sanchis [115] as well as Hendrickson and Leland [61]. The algorithm makes use of $k(k-1)$ priority queues, one for each type of move (source block, target block). A single node movement is done as follows. First, all queues maximizing the gain are found. Then the movement with the highest gain that preserves or improves the balance is performed. Roughly speaking, a node is moved to a block A which maximizes the reduction in the cut when the node is moved. However, the running time of this algorithm is significantly higher than the running time of the two-way Fiduccia-Mattheyses algorithm.

Karypis and Kumar [78] present a k-way version of the Fiduccia-Mattheyses algorithm that runs in linear time $\mathrm{O}(m)$. Instead of $k(k-1)$ priority queues, Karypis and Kumar use one global priority queue for all types of moves. The priority used is the maximum local gain, i.e. the maximum reduction in the cut when the node is moved to one of its neighboring blocks. The node that is selected for movement yields the maximum improvement for the objective and maintains or improves upon the balance constraint. To improve the running time, the priority queue only contains boundary nodes, i.e. nodes that have an external degree greater than zero, and local search is stopped after x movements that did not decrease the overall cut. A more expensive k-way local search algorithm is based on tabu search [55, 56], which has been applied to graph partitioning by [112, 14, 12, 13, 52]. We briefly outline the method reported by Galinier et al. [52]. Instead of moving a node

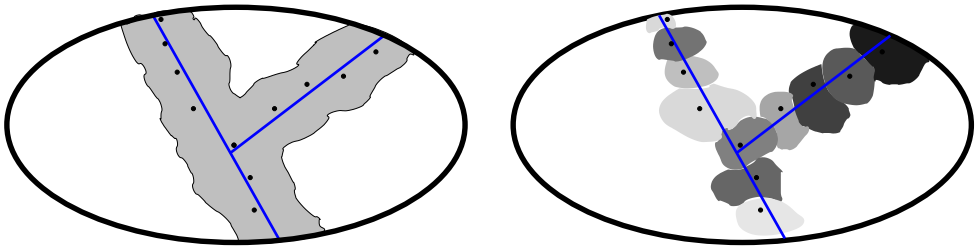


Figure 6.6: On the left hand side a typical k-way local search that is initialized with all boundary nodes is illustrated. On the right hand side Multi-try FM is shown. Multiple k-way searches that are initialized with single boundary nodes are started. In both cases, each node is moved at most once during a round. Source: [117].

only once per pass, as in the traditional versions of the Kernighan-Lin/Fiduccia-Mattheyses algorithms, specific types of moves are excluded only for a number of iterations. The number of iterations that a move (v, block) is excluded depends on an aperiodic function f and the current iteration i. The algorithm always moves a non-excluded node with the highest gain. If the node is in block A, then the move (v, A) is excluded for $f(i)$ iterations after the node is moved to the block yielding the highest gain, i.e. the node cannot be put back to block A for $f(i)$ iterations.

Localization

Results in KaSPar [102] and KaPPa [66] indicate that localization of local search yields increased partition quality. Hence, a localized variant of the k-way local search algorithm has been described in [117]. Previous k-way methods were initialized with *all* boundary nodes, i.e. all boundary nodes are eligible for movement at the beginning. In contrast, the method is repeatedly initialized with a *single* node. Intuitively, this introduces a larger amount of diversification compared to the classical method begin restricted to move a boundary node having the largest gain.

Multi-try FM is organized in rounds. A round works as follows. Instead of putting *all* boundary nodes directly into a priority queue, we put the boundary nodes of the current block pair under consideration into a todo list T. Subsequently, we begin a k-way local search starting with a *single* random node v of T. However, local search is only started if v was not touched by a previous localized k-way search in this round. Either way, v is removed from the todo list. We stress that a localized k-way search is restricted to the movement of nodes that have not been touched by a previous local search during the round. This assures that at most n nodes are touched during a round of the algorithm and that the algorithm can be implemented in linear time. Figure 6.6 illustrates the differences to the global k-local search algorithm.

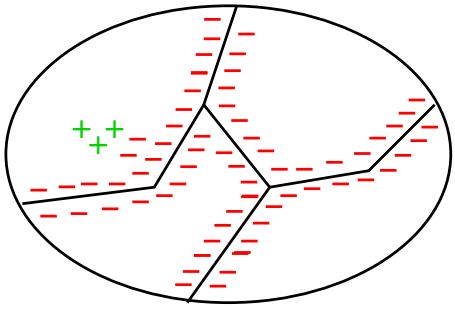


Figure 6.7: An example illustrating why localization helps to find better partitions. – stands for a node with negative gain, + stands for a node with positive gain. Source: [103].

To understand why localization of local search is helpful to create partitions of higher quality, consider the example depicted in Figure 6.7. We see a 4-partition of a graph that is a local optimum in the sense that at least two node movements are necessary until moving a node with positive gain is possible. Recall that classical k-way local search is initialized with all boundary nodes (in this case all of them have negative gain). It then starts to move nodes with negative gain at multiple places of the graph. When it finally moves nodes with positive gain (+) the partition is already much worse than the input partition. Hence, the movement of the positive gain nodes does not yield an improvement with respect to the given input partition. On the other hand, a localized local search that starts close to the nodes with positive gain, can find the positive gain nodes by moving only a small number of nodes with negative gain. Since it did not move as many negative gain nodes as the classical k-way search, it may still finds an improvement with respect to the input partition.

6.4 Think Globally, Act Locally

We now look at a local improvement scheme for the *perfectly* balanced graph partitioning problem. This scheme encodes local searches that are not restricted to a balance constraint into a model allowing us to find combinations of these searches maintaining balance by applying a negative cycle detection algorithm. From a meta heuristic point of view the proposed algorithms increase the neighborhood of a perfectly balanced solution in which local search is able to find better solutions. Moreover, we will see efficient ways to explore this neighborhood.

Roughly speaking, all the following algorithms consist of two components. The *first component* are local searches on pairs of blocks that share a non-empty boundary, i.e. all edges in the quotient graph. These local searches are not restricted to the balance constraint of the graph partitioning problem and are undone after they have been performed. The *second component* uses the information gathered in the first component. That means we build a model using the node movements performed in the first step enabling us to find combinations of those node movements that *maintain balance*.

We begin by describing the very basic algorithm and go on by presenting an advanced model which enables us to combine complex local searches. This is followed by a description on how local search and balancing algorithms are put together.

Basic Idea – Using A Negative Cycle Detection Algorithm

We start with a very simple case where the first component only moves single nodes. A node in the graph G can have two states *marked* and *unmarked*. By default a node is unmarked. It is called *eligible* if it is not adjacent to a previously marked node. We now build the model of the underlying partition of the graph G, $\mathcal{Q} = (\{1, \cdots, k\}, \mathcal{E})$ where $(A, B) \in \mathcal{E}$ if there is an edge in G that runs between the blocks A and B. We define edge weights $\omega_{\mathcal{Q}} : \mathcal{E} \to \mathbb{R}$ in the following way: for each *directed* edge $e = (A, B) \in \mathcal{E}$ in a random order, find a *eligible* boundary node v in block A having maximum gain $g_{\max}(A, B)$, i.e. a node v that maximizes the reduction in cut size when moving it from block A to block B. If there is more than one such node, we break ties randomly. Node v is then marked. The weight of e is then $\omega_{\mathcal{Q}}(e) := -g_{\max}(A, B)$, i.e., the negative gain value associated with moving v from A to B. Note that, in general, $\omega_{\mathcal{Q}}((A, B)) \neq \omega_{\mathcal{Q}}((B, A))$. An example for this basic model is shown in Figure 6.8. Observe that the basic model is a directed and weighted version of the quotient graph and that the selected nodes form an independent set. Note that each cycle in this model defines a set of node movements and furthermore when the associated nodes of a cycle are moved, then

Algorithm 5 Building the model

```
unmark all v ∈ V
compute directed quotient graph Q
for each edge (A, B) ∈ Q in random order do
    find an unmarked node v in block A maximizing gain g_{A,B}
    mark v and all its neighbors
    set the weight of (A, B) to −g_{A,B}(v)
end for
```

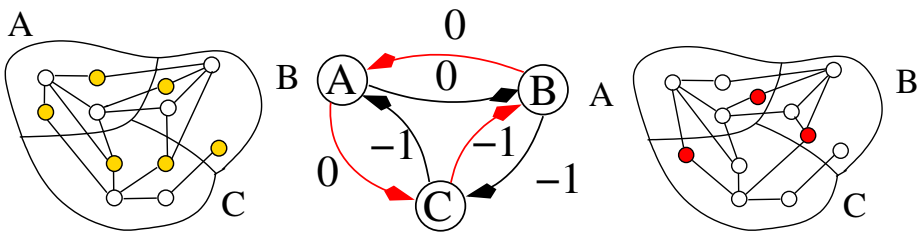


Figure 6.8: Left: example graph partitioned into three parts (A, B and C). Possible candidates are highlighted. Middle: corresponding model and one negative cycle is highlighted. Right: updated partition after associated node movements of cycle are performed. Moved nodes are highlighted.

each block contains the same number of nodes as before. Also the weight of a cycle in the model is equal to the reduction in the cut when the associated node movements are performed. However, the most important aspect is that a *negative cycle* in the model corresponds to a set of node movements that will decrease the overall cut and maintain the balance of the partition. To detect a negative cycle in this model we introduce a node s and connect it to all nodes in $\mathcal{Q}$. The weight of the inserted edges is set to zero. We can apply a standard shortest path algorithm [34] that can handle negative edge weights to detect a negative cycle. If the model contains a negative cycle we can perform a set of node movements that will not alter the balance of the blocks since each block obtains and emits a node. We can find additional useful augmentations by connecting underloaded blocks to s by a zero weight edge. Now, negative cycles containing s change some block weights but will not violate any additional balance constraints. Indeed, when the node following s is overloaded initially, this overload will be reduced.

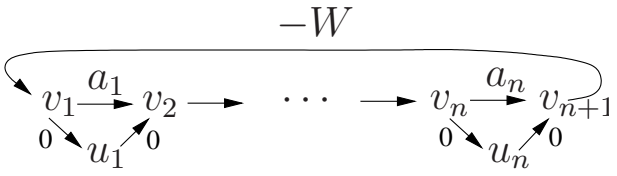
If there is no negative cycle in the model, we apply a diversification strategy based on cycles of weight zero. This strategy is explained below. An interesting observation is that the algorithm can be seen as an extension of the classical FM algorithm [46] which swaps nodes between two adjacent blocks (two at a time) which is basically a negative cycle of length two in our model if the gain of the two node movements is positive.

Diversification by Zero Gain Cycle Moves

A zero weight cycle in the basic model is associated with a set of node movements that keep the cut unchanged and the block weights constant. After such a movement is performed it might be possible to find further negative cycles in the basic model since candidates for movements and gain values may have changed. Hence these cycles can be useful to introduce some diversification.

Theorem 8
Given a directed graph G with arbitrary edge weights $\omega : E \to \mathbb{R}$, the decision problem that outputs yes if there exists a zero-weight cycle in that graph and no otherwise, is NP-complete.

Proof. We reduce the subset sum problem to our problem. The subset sum problem is stated as follows: given $S = \{a_1, \ldots, a_n\}$ and W, is there a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = W$. To reduce subset sum to our problem, we use the following construction:



It is easy to see that the construction can be done in polynomial time. Moreover, if the subset sum instance is a yes instance, then the constructed graph contains a zero-weight cycle and vice-versa. □

Nonetheless, on general graphs it is NP-complete to decide whether a weighted graph contains a cycle that has weight zero, i.e. the sum of the edge weights of this cycle is zero (see Theorem 8). However, we will see that if a graph does not contain a negative cycle we can decide whether it contains a cycle of weight zero in polynomial time and output one if one exists. This can be done by using the following technique. As soon as the model described above does not contain negative cycles we compute a shortest path tree starting at s. By doing this we obtain node potentials $\Pi : \{1, \cdots, k\} \to \mathbb{R}$, i.e. the shortest path distances from s to all other nodes. We then define modified edge weights $\ell_{\mathcal{Q}}(e = (A, B)) = \omega_{\mathcal{Q}}(e) + \Pi(A) - \Pi(B)$. It is quite easy to see that the weight of a cycle in $\mathcal{Q}$ does not change when we use $\ell_{\mathcal{Q}}$ instead of $\omega_{\mathcal{Q}}$. In particular cycles that have weight zero w.r.t $\omega_{\mathcal{Q}}$ will have weight zero w.r.t. $\ell_{\mathcal{Q}}$. Another important observation is that $\ell_{\mathcal{Q}}$ is a non-negative function. Hence, in order to detect a zero weight cycle we can evict all edges e with $\ell_{\mathcal{Q}}(e) > 0$ since they cannot be a part of a cycle having weight zero. After this is done we compute all strongly connected components of this graph. If there is a strongly connected component that contains more than one node then the graph contains a cycle that has weight zero. To output one zero weight cycle we pick a random node N out of the components having more than one node. Starting at this node we perform a random walk in its component until we find a node that we have already seen M (which is not necessary N). It is then fairly simple to output the respective cycle starting at M. Note that if the component contains j nodes than the random walk will stop after at

Algorithm 6 Finding zero weight cycles in graphs without negative cycles

insert a node s and edges $(s, B)\ \forall v \in V_{\mathcal{Q}}$
compute the shortest-path tree starting at s
define $\Pi : V_{\mathcal{Q}} \to R$ as the shortest-path distances
$l(A, B) := w(A, B) + \Pi(A) - \Pi(B)$ are the new edge weights
delete all edges with weight $l(A, B) \neq 0$
compute the SCCs of that graph
perform a random walk in a SCC with more than nodes to find a cycle

most j iterations. As soon as we have found a cycle of weight zero we can perform the node movements that are associated with the edges of the cycle.

It can be easily seen that the time complexity of the zero weight cycle finding procedure is $O(|V_{\mathcal{Q}}||E_{\mathcal{Q}}|)$. Inserting the node s is $O(|V_{\mathcal{Q}}|)$, since that many edges have to be introduced. The shortest path algorithm takes $O(|V_{\mathcal{Q}}||E_{\mathcal{Q}}|)$. Edge weights can be updated in $O(|E_{\mathcal{Q}}|)$, along with directly removing edges. SCCs can be computed in $O(|V_{\mathcal{Q}}| + |E_{\mathcal{Q}}|)$ by depth first search. Finally, the random walk needs $O(|SCC|) \in O(|V_{\mathcal{Q}}|)$, where $|SCC|$ is the number of nodes of the SCC.

Advanced Model

We now integrate advanced local search algorithms. Each edge in the advanced model stands for a *set* of node movements found by a local search. Hence, a negative cycle corresponds to a combination of local searches with positive overall gain that maintain balance or that can improve balance. Before we build the advanced model we perform *directed local search* on each pair of blocks that share a non-empty boundary, i.e. each pair of blocks that is adjacent in the quotient graph. A local search on a directed pair of blocks (A, B) is only allowed to move nodes from block A to block B. The order in which the directed local search between a directed pair of blocks is performed is random. That means we pick a random directed adjacent pair of blocks on which local search has not been performed yet and perform local search as described below. This is done until local search was done between all directed adjacent pairs of blocks once.

Directed Local Search

We now explain how we perform a directed local search between a pair (A, B) of blocks. A directed local search between two blocks A and B is very localized akin to the multi-try method used in KaFFPa [117]. However, a directed local search between A and B is restricted to move nodes from block A to block B. It is similar to the FM-algorithm: We start with a *single* random eligible boundary

node of block A having maximum gain $g_{\max}(A, B)$ and put this node into a priority queue. The priority queue contains nodes of the block A that are valid to move. The priority is based on the *gain*, i.e. the decrease in edge cut when the node is moved from block A to block B. We always move the node that has the highest priority to block B. After a node is moved, its eligible neighbors that are in block A, are inserted into the priority queue. We perform at most τ steps per directed local search, where τ is a parameter. Note that during a directed local search we only move nodes that are not incident to a node moved during a previous directed local search. This restriction is necessary to keep the model described below accurate. Thus we *mark all nodes* touched during a directed local search *after* it was performed which also implies that each node is moved at most once. In addition, all moved nodes are *moved back* to their origin, since these movements would make the partition imbalanced. We stress that all nodes incident to nodes that have been moved during a directed local search are not *eligible* for any later local search during the construction since this would make the gain values computed imprecise.

The Model Graph

The advanced model allows us to find combinations of directed local searches such that the balance of the given partition is at least maintained. The challenge here is that, in contrast to movements of single nodes, we cannot combine arbitrary local searches since they do not all move the same number of nodes. Hence, we specify a more sophisticated graph with the property that a negative cycle maintains feasibility.

The local search process described above yields for each pair of blocks $e = (A, B)$ in the quotient graph a sequence of node movements S_e and a sequence of gain values g_e. The d'th value in g_e corresponds to the reduction in the cut between the pair of blocks (A, B) when the first d nodes in S_e are moved from their source block A to their target block B. By construction, a node $v \in V$ can occur in at most one of the sequences created and in its sequence only once.

Generally speaking, the *advanced model* consists of τ layers. Essentially each layer is a copy of the quotient graph. An edge starting and ending in layer d of this model corresponds to the movement of exactly d nodes. The weight of an edge $e = (A, B)$ in layer d of the model is set to the negative value of the d'th entry in g_e. In other words, it encodes the negative value of the gain, when the first d nodes in S_e are moved from block A to block B. Hence, a negative cycle whose nodes are all in layer d will move exactly d nodes between each of the respective block pairs contained in the cycle and results in a overall decrease in the edge cut. We add additional edges to the model such that it contains *more possibilities* in presence of underloaded blocks. To be more precise, in these cases we want to get

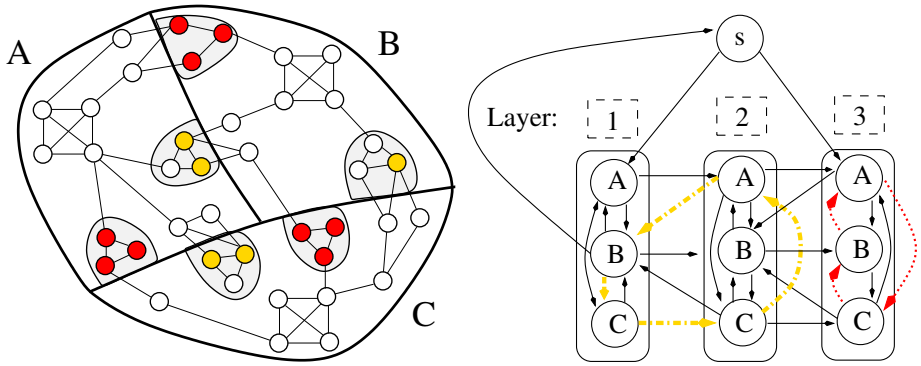


Figure 6.9: On top a graph that is partitioned into three parts ($|A| = 14, |B| = 12, |C| = 14$). Directed local searches on each directed pair of blocks are highlighted ($\tau = 3$). The corresponding advanced model is shown on the bottom. Each layer is a copy of the quotient graph of the partition. Edges within layer d represent node movements consisting of d nodes that have been found previously using directed local search. s is connected to all nodes (most of the edges are not shown), edges back to s are inserted if the corresponding block can take some nodes without becoming overloaded (in this example block B), backward edges between layers are inserted if the block can take nodes without becoming overloaded, forward edges between the layers are inserted in any case. Within layer 3 a negative cycle is highlighted (red/dark dashed) which corresponds the movement of the nine red/dark nodes on top. Another negative cycle is highlighted in yellow/light grey dashed. It corresponds to the movement of the five yellow/light gray nodes on top. The weight of both cycles is -2. After these movements are performed the resulting partition is perfectly balanced.

rid of the restriction that each block sends and emits the same number of nodes. To do so we insert *forward* edges between all consecutive layers, i.e. block k in layer d is connected by an edge of weight zero to block k in layer $d + 1$. These edges are not associated with node movements. Furthermore, we add *backward* edges as follows: for an edge (A, B) in layer d we add an edge with the same weight between block A in layer d and block B in layer $d - \ell$ if block B can take ℓ nodes without becoming overloaded. The newly inserted edge is associated with the same node movements as the initial edge (A, B) within layer d. This way we encode movements in the model where a block can emit more nodes then it gets and vice versa without violating the balance constraint. Additionally we connect each node in layer d back to s if the associated block can take at least d nodes without becoming overloaded. Again this means that the model might contain cycles through s which stand for paths in the quotient graph being associated with node movements that decrease the overall cut. Moreover, these moves never

increase the imbalance of the input partition. An example for the advanced model can be found in Figure 6.9. We can apply the same zero weight cycle diversification as in the basic model. The advanced model can contain *conflicting* cycles that cannot be used. Due to space constraints, we explain when conflicts occur and how we handle them in the TR [119].

Multiple Directed Local Searches

The algorithm can be further improved by performing multiple directed local searches (MDLS) between each pair of blocks that share a non-empty boundary. More precisely, after we have computed node movements on *each* pair of blocks $e = (A, B)$, we start again using the nodes that are still eligible. This is done μ times. The model is then slightly modified in the following way: For the creation of edges in the model that correspond to the movement of d nodes from block A to block B we use the directed local search on $e = (A, B)$ from the process above with the best gain when moving d nodes from block A to block B (and use this gain value for the computation of the weight of corresponding edges).

Balancing

As we will see, to create ϵ-balanced partitions we start our algorithm with partitions where larger imbalance is allowed. Hence, to satisfy the balance constraint, we have to think about balancing strategies. A balancing step will only be applied if the model does not contain a negative cycle (see next section for more details). Hence, we can modify the advanced model such that we can find a set of node movements that will decrease the total number of overloaded nodes by at least one and minimizes the increase in the number of edges cut. Specifically, we introduce a second node t. Now instead of connecting s to all vertices, we connect it only to nodes representing overloaded blocks, i.e. $|V_i| > \lceil |V|/k \rceil$. Additionally, we connect a node in layer ℓ to t if the associated block can take at least ℓ nodes without becoming overloaded. Since the underlying model does not contain negative cycles, we can apply a shortest path algorithm to find a shortest path from s to t. We use a variant of the algorithm of Bellman and Ford since edge weights might still be negative It is now easy to see that a shortest path in this model yields a set of node movements with the smallest increase in the number of cut edges and that the total number of overloaded nodes decreases by at least one. If τ is set to one we call this algorithm basic balancing otherwise advanced balancing.

However, we have to make sure that there is at least one s-t path in the model. Let us assume for now that the graph is connected. If the graph is connected then the directed version of the quotient graph is strongly connected. Hence an s-t path exists in the model if we are able to perform local search between *all*

pairs of blocks that share a non-empty boundary. Because a directed local search can only start from an eligible node, we might not be able to perform directed local search between all adjacent pairs of blocks, e.g. if there is no eligible node between a pair of blocks left. We try to *ensure* that there is at least one s-t path in the model by doing the following. Roughly speaking we try to integrate an s-t path into the model by changing the order in which directed local searches are performed. First we perform a breadth first search (BFS) in the quotient graph which is initialized with all nodes that correspond to overloaded blocks in a random order. We then pick a random node in the quotient graph that corresponds to a block A that can take nodes without becoming overloaded. Using the BFS-forest we find a path $\mathcal{P} = B \to \cdots \to A$ from an overloaded block B to A.

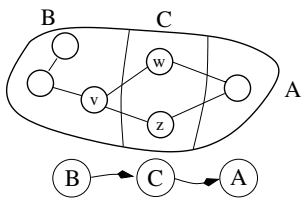


Figure 6.10: Top: a graph partitioned into three parts. Bottom: BFS-tree in the quotient graph starting in overloaded block B. This path cannot be integrated into the model. After a directed local search on pair (B, C), v is marked and there is no eligible node left for the local search on pair (C, A). A similar argument holds if local search is done on the pair (C, A) first.

We now first perform directed local search on all consecutive pairs of blocks in $\mathcal{P}$. Here, we use $\tau = 1$ for the number of node movements to minimize the number of non-eligible nodes. If this was successful, i.e. we have been able to move one node between all directed pairs of blocks in that path, we perform directed local searches as before on *all* pairs of blocks that share a non-empty boundary. Otherwise we undo the searches done (every node is eligible again) and start with the next random block that can take a node without becoming overloaded. In some rare cases the algorithm fails to find such a path, i.e. each time we look at a path we have one directed pair of blocks where no eligible node is left. An example is shown in Figure 6.10. In this case we apply a fallback balance routine that guarantees to reduce the total number of overloaded nodes by one if the input graph is connected. Given the BFS-forest of the quotient graph from above, we look at all paths in it from an overloaded block to a block that can take a node without becoming overloaded. At this point there are at most $\mathrm{O}(k)$ such paths in our BFS-forest. Specifically for a path $\mathcal{P} = Z \to Y \to X \to \cdots \to A$ we select a node having maximum gain $g_{Z,Y}$ in Z and move it to Y. We then look at Y and do the same with respect to X and so on until we move a node to block A. Note that this time we can ensure to find nodes because after a node has been moved it is not blocked for later movements. After the operations have been performed they are undone and we continue with the next path. In the end we use the movements of the path that resulted in the smallest number of edges cut. If the graph contains more than one connected component then the algorithms described above may not work (see [119]).

Putting Things Together

In practice we start our algorithms with an unbalanced input partition. We define two algorithms, basic and advanced, depending on the models used. Both the basic and the advanced algorithm operate in rounds. In each round we iterate the negative cycle based local search algorithm until there are no negative cycles in the corresponding model (basic or advanced). After each negative cycle local search step we try to find zero weight cycles in the model to introduce some diversification. Since we have random tie breaking at multiple places, we iterate this part of the algorithm. If we do not succeed to find an improved cut using these two operations for λ iterations, we perform a single balancing step if the partition is still unbalanced; otherwise we stop. The parameter λ basically controls how fast the unbalanced input partition is transformed into a partition that satisfies the balance constraint. After the balancing operation, the total number of overloaded nodes is reduced by at least one depending on the balancing model. In the basic algorithm we use the basic balancing model ($\tau = 1$) and in the advanced algorithm we use the advanced balancing model. Since the balance operation can introduce new negative cycles in the model we start the next round.

6.5 Maximum Flows as Local Search

Ford and Fulkerson [49] presented their well-known max-flow min-cut theorem in 1956. While it can be used to separate two nodes in a graph by computing a maximum flow and hence a minimum cut between them, it completely ignores balance, and it is unclear how it could directly be applied to the balanced graph partitioning problem. However, the algorithm is often used as a subroutine to solve related max-flow problems, e.g. to bisect regular graphs by Bui et al. [28], to improve a given partition when quality is measured by expansion or conductance by Lang and Rao [88] and Andersen and Lang [4], or as a pre-processing technique for road network partitioning by Delling et al. [39]. Note that the problem that asks to find a non-trivial cut (V_1, V_2) with minimum conductance or expansion, does not necessarily yield a balanced cut as in the balanced graph partitioning problem.

6.5.1 Maximum Flows to Improve Edge Cut

We start to explain how maximum flows can be employed to improve a balanced partition of *two blocks* V_1, V_2 without violating the balance constraint. Informally speaking, we want to find an area around the initial cut of the partition such that each s-t cut in this area corresponds to a balanced cut of the input graph. That yields a local improvement algorithm which can be used in a multilevel framework.

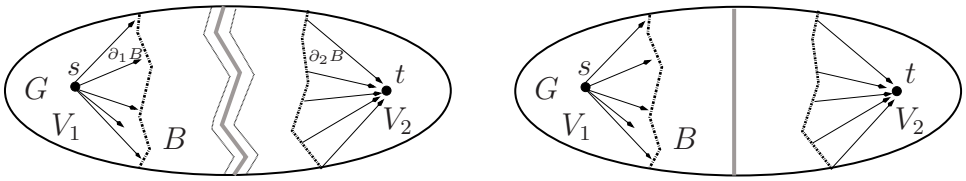


Figure 6.11: The construction of a flow problem $\mathcal{F}$ is shown on the left hand side. Each cut within B induces a cut within the balance constraint in the original graph. A min cut in this area is shown on the right hand side. Source: [117].

First we introduce a few notations. Given a set of nodes $B \subset V$, we define its *border* $\partial B := \{u \in B \mid \exists (u,v) \in E : v \notin B\}$. The set $\partial_1 B := \partial B \cap V_1$ is called *left border* of B and the set $\partial_2 B := \partial B \cap V_2$ is called *right border* of B. A B *induced flow problem* $\mathcal{F}$ is the node induced subgraph $G[B]$ using the edge weights of G as capacities, plus two nodes s, t that are connected to the border of B. More precisely, s is connected to all left border nodes $\partial_1 B$ and all right border nodes $\partial_2 B$ are connected to t. These new edges get the capacity ∞. Note that the additional edges are directed. $\mathcal{F}$ has the *cut property* if each (s,t)-cut induces a feasible cut in G.

The basic idea is to construct a flow problem $\mathcal{F}$ having the cut property. Each min-cut will then yield a feasible improved cut within the balance constraint in G. By performing two breadth-first searches we can find such a set B. Each node touched during these searches belongs to B. The first breadth-first search is initialized with the boundary nodes in V_1 and is only expanded into V_1. As soon as the weight of the area found by this breadth-first search would exceed $(1+\epsilon)c(V)/2 - c(V_2)$, we stop the search. The second breadth-first search is done analogously for V_2. The constructed flow problem has the cut property since the worst case new weight of V_2 is lower or equal to $c(V_2) + (1+\epsilon)c(V)/2 - c(V_2) = (1+\epsilon)c(V)/2$. The same holds for the worst case new weight of V_1. Figure 6.11 explains the construction, an example is shown in Figure 6.12.

There are multiple ways to *improve* this method. This includes *repeated usage* of the method, larger areas B in which the flow problem is solved and the search for better balanced cuts using the information already provided by the max-flow computation. First, if we found an improved cut, we can apply the method again since the initial boundary has changed, i.e. the set B will also have changed. Secondly, we can *adaptively control the size* of the flow problem by dropping the feasibility constraint. This enables us to search for cuts that fulfill the balance constraint in a *larger subgraph*. If the found min-cut in $\mathcal{F}'$ (using $\epsilon' = \alpha\epsilon$ with $\alpha > 1$ for construction) fulfills the balance constraint in G, we accept it and increase α to $\min(2\alpha, \alpha')$ where α' is an upper bound for α. Otherwise, the cut

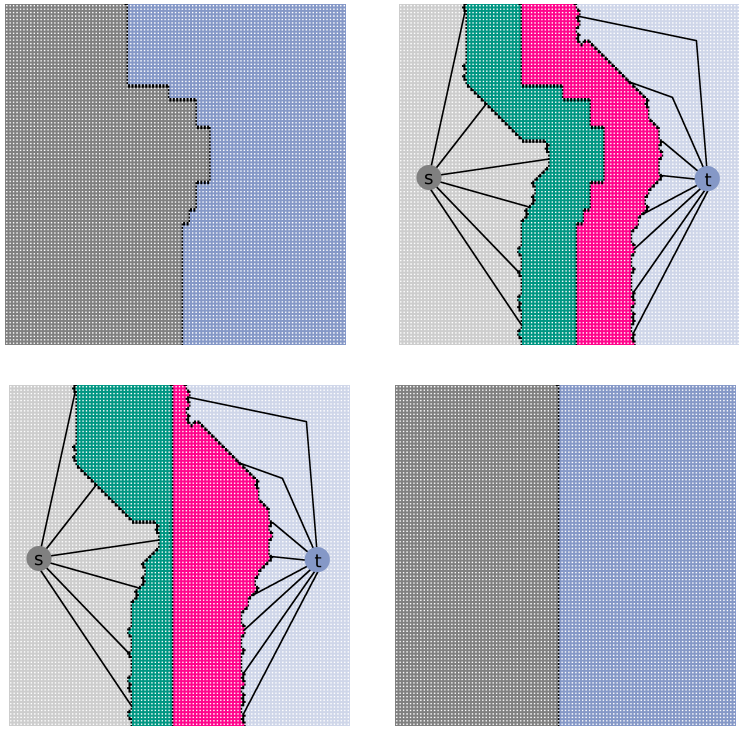


Figure 6.12: Top left: An example grid that is partitioned into two blocks. Top right: The constructed flow problem. Bottom left: The minimum cut in this area. Bottom right: The final partition corresponding to the minimum cut of the flow problem. Source: [117].

is not accepted and we decrease α to $\max(\frac{\alpha}{2}, 1)$. This method is iterated until a maximal number of iterations is reached or if the computed cut yields a feasible partition without a decreased cut. We call this method *adaptive flow iterations*.

Most Balanced Minimum Cuts

The third idea to improve this method is to use the information already provided by the max-flow computation to extract a cut that has better balance. Picard and Queyranne [108] have shown that *one* (s, t)-max-flow contains information about *all* minimum (s,t)-cuts in the graph. We present a heuristic that, given a max-flow, aims to output a better balanced minimum cut with respect to the balance of the induced bipartition in G. Having an algorithm at hand that not only outputs one minimum cut but also a cut with good balance makes the idea to search for feasible cuts in larger subgraphs even more attractive. Recall, for a graph $G = (V, E)$ a set $C \subseteq V$ is a *closed node set* iff for all nodes $u, v \in V$, the conditions $u \in C$ and $(u, v) \in E$ imply $v \in C$. In other words, there is no edge starting in C and ending

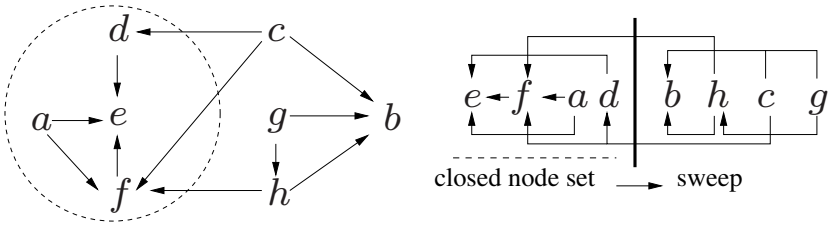


Figure 6.13: Left: the set $C = \{a, d, e, f\}$ is a closed node set since no edge is starting in C and ending in $V \backslash C$. Right: using a reverse topological ordering of a DAG one can output multiple closed node sets. Source: [117].

in $V \backslash C$. An example can be found in Figure 6.13. We now formulate the Lemma of Picard and Queyranne:

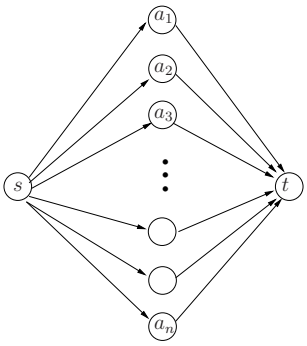
Lemma 9 (Picard and Queyranne [108])
Each closed node set containing s in the residual graph of a maximum (s, t)-flow yields a minimum (s, t)-cut.

Specifically, given a closed node set C of the residual graph containing s, the corresponding min-cut is $(C, V \backslash C)$. Note that distinct maximum flows may produce different residual graphs but the closed node sets remain the same. We now show how the residual graph can be compactified and explain our heuristic to compute closed node sets inducing better balanced min-cuts with respect to balance of the bipartition in the original graph.

Observe that a cycle in the residual graph cannot contain a node of both, a closed node set and its complement. Hence, to enumerate all minimum cuts of a graph, Picard and Queyranne [108] contract the strongly connected components of the residual graph which results in a weighted, directed acyclic graph (DAG). We refer to this DAG as *minimum cut representation*. That the problem of finding the most balanced minimum cut is still NP-hard has been shown by Bonsma [22] (see Theorem 10).

Theorem 10
Given a s-t flow network. the problem of finding the min cut $(S, V \backslash S)$ that maximizes $\min(|S|, |V \backslash S|)$ is NP-complete [22].

Proof. We reduce the partition problem to our problem. The partition problem is stated as follows: given $\{a_1, \ldots, a_n\}$, is there a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$. The construction is as follows:



The construction can clearly be done in polynomial time. Note that all non-trivial cuts $(S, V \backslash S)$ cut n edges and if the partition instance is a yes instance, then the most balanced minimum cut is perfectly balanced and vice versa. □

Note that each closed node set of the minimum cut representation induces a minimum cut and that we still can compute all minimum cuts using the compactified representation. Hence, we can search for closed node sets containing the component S that contains the source s in the minimum cut representation. We do that by using the following heuristic which is repeated a few times using different random seeds for initialization.

The basic idea is that using a topological order yields closed node sets quite easily. Therefore, we first compute a random topological order by using a randomized depth first search. We sweep through the reverse topological order and sequentially add the components to the closed node set. By doing so we compute closed node sets each inducing a min-cut having a different balance. We use the best balanced minimum cut with respect to the original bipartition found. The closed node set with the best balance occurred using different random topological orders is returned by the algorithm. Note that this procedure can still find cuts that are not feasible in oversized subgraphs, e.g. if there is no feasible minimum cut. Therefore the algorithm is combined with the adaptive strategy from above. We call this method *balanced adaptive flow iterations*.

Local search and maximum flows can be compared and characterized by their properties. Local search has the disadvantage of making only local improvements possible, whereas maximum flows can result in more global improvements. In practice, however, a combination of both approaches works best.

6.5.2 Max-flow Quotient-cut Improvement

Max-flow Quotient-cut Improvement (MQI) by Lang and Rao [88] and Improve by Andersen and Lang [4] are flow-based methods for improving graph bipartitions when cut quality is measured by quotient-style metrics such as expansion or conductance. In this case, the balance constraint is dropped, since the measures directly optimize cut vs. balance of the partition. Given a partition V_1, V_2 of the graph, MQI constructs a flow problem such that the output partition is the best improvement among all partitions, where V_1' is a strict subset of V_1 w.r.t. the quotient-style metric. The flow problem is constructed as follows. Let V_1 be the smaller block, c be the number of cut edges and a the number of nodes in V_1. First, the construction completely discards all nodes in V_2. Each undirected edge in V_1 is replaced by two directed edges with capacity a, one in each direction. Then a source and a sink are inserted. Each node in V_1 is connected to the sink by a directed edge with capacity c and the source is connected to all boundary nodes in V_1 via a directed edge with capacity a. Figure 6.14 illustrates the construction and Figure 6.15 shows a real world example. Solving this flow problem, the authors can construct a new partition having best quotient cut score among all partitions (V_1', V_2'), where V_1' is a strict subset of V_1 if one exists. Improve [4] solves a slightly different flow problem that takes both blocks into account and is able to always outperform or tie MQI. Since both approaches need a pre-partition of the graph, the authors combine their algorithms with Metis [77] and have been able to improve some of the bipartitions in the Walshaw Benchmark.

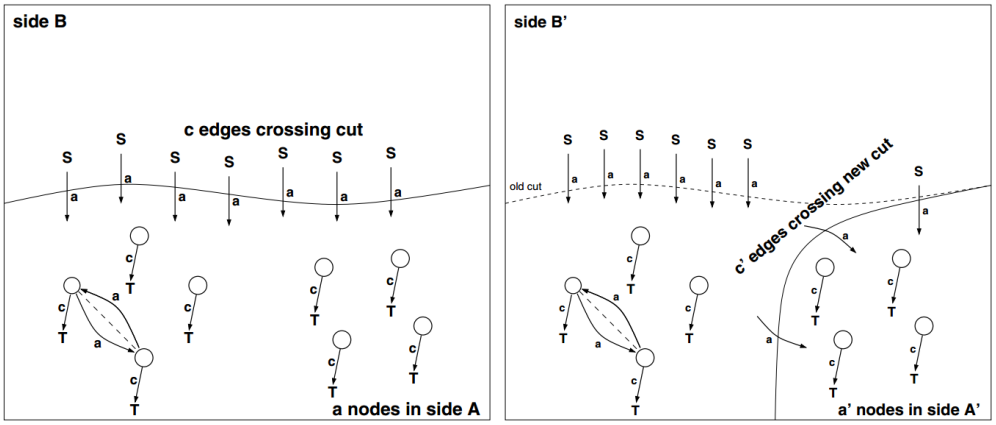


Figure 6.14: Visualization of the MQI construction.

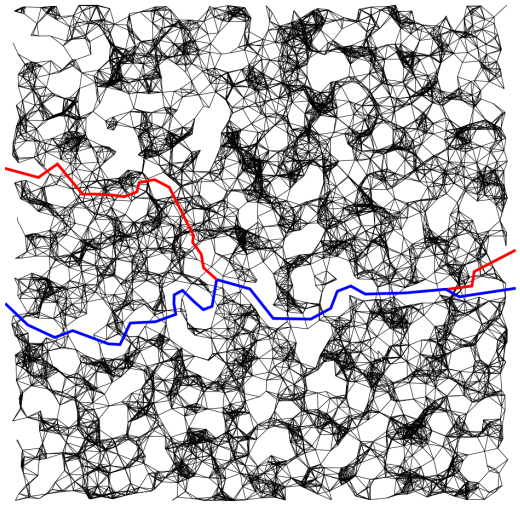


Figure 6.15: A real world graph cut into two partitions. The red cut is produced by METIS and the improved blue cut is produced by METIS+MQI which uses the algorithm discussed in this chapter.

Theorem 11
There is an improved quotient cut (A', B') *(with* $A' \subset A$*), iff the maximum flow is less than* ca*, with* $a = \min\{|A|, |B|\}$ *and* $c = |E(A, B)|$.

Proof. $\Rightarrow$: We start by assuming the existence of an improved quotient cut (A', B'), with A' a strict subset of A. Let us say that A' contains a' nodes and that c' edges cross the improved cut. Then by the definition of quotient cuts, $\frac{c'}{a'} < \frac{c}{a}$. The net flow into A' on non-sink edges is at most $c'a$, but a flow of $a'c$ would be required to saturate the a' sink edges of A'. The above inequality implies that $c'a < a'c$, so insufficient flow is available, and some sink edge leading out of A' must be unsaturated. But if any sink edge is unsaturated, the total flow into the sink must be less than ca, and our claim is proved.
$\Leftarrow$: Here we assume that the maximum flow is less than ca. We will construct a new cut and prove that it has an improved quotient cut score. The small side A' of the new cut is the set of nodes reachable from the sink by a depth-first search along reversed unsaturated edges. This "backwards DFS" procedure advances from a node y to a node x if and only if there is an unsaturated edge leading from node x to node y. The analysis of the new cut begins by noting that the total capacity of all sink edges is ac, while the total flow is less than ca, so at least one sink edge must be unsaturated. Therefore at least one node of A is reachable by the backwards DFS, so A' is nonempty, and we will be able to divide by $a' = |A'|$ a little later in the proof. Also, the existence of at least one unsaturated edge leading from A' to the sink means that F_s, the total flow from A' to the sink, must be less than $a'c$.

Now, since we built A' by going as far back as we could along unsaturated edges, every edge feeding into A' from a non-A' node must be saturated. Let us say there are c' such edges. Then F_i , the flow into A', is exactly $c'a$.

Recall that the flow is in a canonical form with no flow contrary to any saturated edge. Therefore F_o , the flow out of A', is entirely carried by sink edges, so $F_o = F_s$. Finally, by conservation of flow, $F_i = F_o$. Combining these facts, $c'a = F_i = F_o = F_s < a'c$. Because $a \geq 1$ and $a' >= 1$, we can divide to get $\frac{c'}{a'} < \frac{c}{a}$, which proves that (A', B') is a better quotient cut than (A, B).

References

This chapter is based on Kernighan and Lin [79], Fiduccia and Mattheyses [46], Lang and Rao[88] and Schulz [122], where some passages were taken from. It was created by Demian Hespe, Sebastian Bayer, Eike Röhrs and Christian Schulz.

Chapter 7

Initial Partitioning

Their are many methods to obtain a partition from a graph. For example we could use techniques that we already explored such as spectral partitioning or even solving the problem exact using the integer linear programming approach. Another commonly used method to obtain a bisection of the coarsest graph is graph growing [77]. Its simplest version works as follows. Starting from a random node v, the blocks are assigned using a breadth-first search starting at v. All nodes touched during the breadth-first search are assigned to block V_1. The search is stopped after half of the original node weights are assigned to this block and V_2 is set to $V \backslash V_1$. The computed partitions have a rather large cut because breadth-first search does not care about cut edges at all. Hence, the authors use a local search algorithm to improve the partition. Moreover, the method depends heavily on the chosen start node v, so that the method must be repeated several times to get a good solution.

There are two variations of this algorithm. An algorithm called greedy graph growing [77] takes the resulting cut into account. Instead of performing a simple breadth-first search, the algorithm always adds the node to the block that results in the smallest increase in the cut. This can be implemented similar to the Fiduccia-Mattheyses algorithm, i.e. using the same data structures.

A variant of the algorithm by George et al. [54] first searches for two nodes that are "far" away from each other. Such nodes are called pseudo peripheral nodes. To find such nodes one first chooses a random node v. Starting at v one performs a breadth-first search that explores the whole graph. The last node w touched by the breadth-first search then serves as new start node for the next round. This is repeated a few times. When the process is stopped, one hopes to have two nodes that have a large distance in the graph. These nodes serve as seed nodes for the assignment of blocks. To obtain a partition of the graph, two breadth-first searches, one for each node, are started and performed alternately. Nodes touched by the first breadth-first search are assigned to block one and the nodes touched by the second breadth-first search are assigned to block two. An example is shown in Figure 7.1.

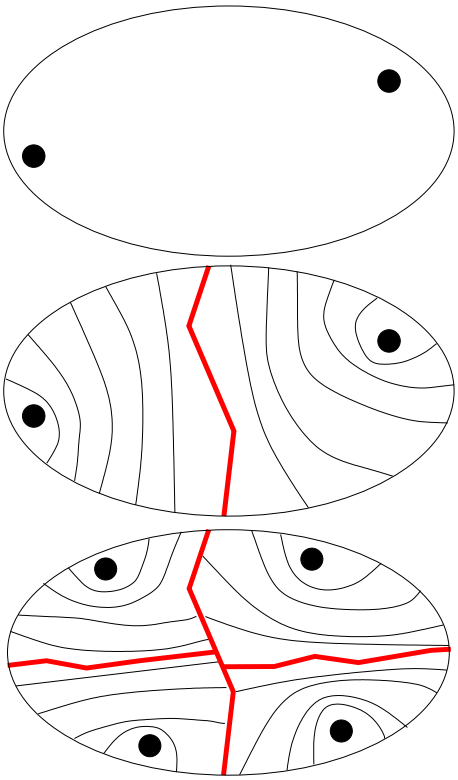


Figure 7.1: Visualisation of graph bisection and recursive bisection. First two pseudoperipheral nodes are found and then two breadth-first search are performed alternately to assign nodes to blocks. Recursive bisection proceeds recursively on each block.

Another variation of the algorithms is to extend the initial partitioning algorithm to be become multilevel algorithms as well. In addition, these algorithms are repeated a couple of times using different random seeds to obtain multiple partitions. The best partition is then selected to be the initial partition of the coarsest graph.

There are also different algorithms to obtain a k-partition of the graph. We shortly discuss recursive bisection. If k is not even, the graphs are split into two blocks, V_1 and V_2, such that $c(V_1) \leq (1+\epsilon)\lfloor \frac{k}{2} \rfloor \lceil \frac{c(V)}{k} \rceil$ and $c(V_2) \leq (1+\epsilon)\lceil \frac{k}{2} \rceil \lceil \frac{c(V)}{k} \rceil$. Block V_1 will be recursively partitioned in $\lfloor \frac{k}{2} \rfloor$ blocks and block V_2 will be recursively partitioned in $\lceil \frac{k}{2} \rceil$ blocks. Note that allowing ϵ imbalance in each bipartition step can result in k-partitions having a larger imbalance. Hence, smaller values of ϵ are usually used during initial partitioning.

Chapter 8

Graph Coarsening

The last chapters discussed several local search and initial partitioning methods. Recall that a multilevel algorithm starts by creating a sequence of smaller graphs. This sequence can be created by different approaches such as contracting edges that form a matching, contracting clusters from a clustering or by techniques similar to algebraic multigrid. We start this chapter by explaining algorithms to find good matchings fast and repeating matching-based contraction. We then finish the chapter with algorithms that contract whole clusterings and algorithms inspired by algebraic multigrid.

8.1 Matching-based Coarsening

Recall the notion of contracting an edge: *Contracting* an edge $\{u, v\}$ means to replace the nodes u and v by a new node x connected to the former neighbors of u and v. We set $c(x) = c(u) + c(v)$ so that the weight of a node at each level is the number of nodes it is representing in the original graph. If replacing edges of the form $\{u, w\}$, $\{v, w\}$ would generate two parallel edges $\{x, w\}$, a single edge with $\omega(\{x, w\}) = \omega(\{u, w\}) + \omega(\{v, w\})$ is inserted. *Uncontracting* an edge e undoes its contraction. In order to avoid tedious notation, G will denote the current state of the graph before and after a (un)contraction unless we explicitly want to refer to different states of the graph. One possibility to select candidates for contraction is to compute a matching in the graph and then contract every edge contained in the matching.

Matching. A *matching* $\mathcal{M}$ is a set of edges not sharing any end point. That is $G = (V, \mathcal{M})$ has maximum degree one. A matching is *maximal*, iff no edge can be added to the matching. The *weight* of a matching is $\sum_{e \in \mathcal{M}} w(e)$. A *maximum weight matching* has the largest possible weight of all matchings of the given graph.

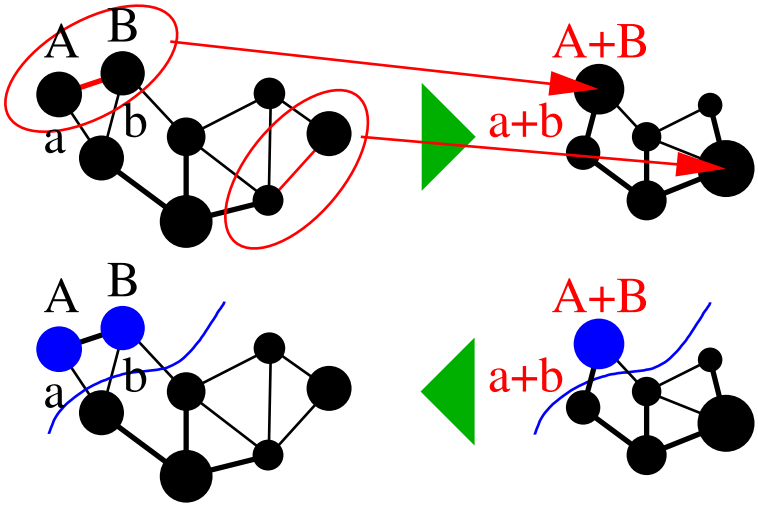


Figure 8.1: A example matching highlighted in red and contraction of matched edges. After a partition has been computed on the coarser graph, it can be transferred to the finer level by putting the finer nodes into the partition of their coarse representative. Due to the way the contraction is defined, edge cut and balance are the same after solution transfer. Source: [122].

Given a matching $\mathcal{M}$ for a weighted graph we can create a new graph by merging all of the pairs of nodes adjacent to an edge $e = \{u, v\} \in \mathcal{M}$. The weight of the new node is equivalent to the weights of the two merged nodes. We transfer all edges to the new graph. If this results in multi-edges we combine those into a single edge and assign its weight as the sum of the weight of the original edges. The result of such a transformation can be seen in Figure 8.1. Note that balance and cut are equal on both sides of this transformation. This means that moving merged nodes is the same as moving a set of nodes in the original graph.

8.2 Matching Algorithms

We now discuss algorithms to compute a matching in a graph. Intuitively, a matching should contain *large edge weights* since we want to solve the problem on the coarsest level and our main objective is to find a small cut. On the other hand a matching should contain a *large number of edges*, e.g. being maximal, so that there are only few levels in the hierarchy and the algorithm can converge quickly. Since our objective is to partition very large graphs, finding an optimal matching takes too much time (see Theorem 12). In practice one relies on heuristics.

Algorithm 7 Greedy Algorithm

$\mathcal{M}_{Greedy} := \emptyset$
while $E \neq \emptyset$ **do**
  take an edge $\{v, w\} \in E$ with *highest* weight
  add $\{v, w\}$ to $\mathcal{M}_{Greedy}$
  remove all edges incident to v or w from E
end while

Theorem 12 (Gabow [51])
A maximum weight matching for a weighted graph can be found in time $O(nm + n^2 \log n)$.

Greedy Algorithm. The algorithm starts by sorting all edges by descending weight and after that add the edge with the highest weight to the matching. All adjacent edges are removed from the graph. The process is repeated with the heaviest edge in the remaining graph until no more edges are left. Since we remove all adjacent edges after adding an edge to the matching the set of edges returned fulfills the matching property. The resulting algorithm can be found in Algorithm 7. The running time of this algorithm is dominated by the time it takes to sort the edges by weight and thus is $O(m \log n)$ (why $\log n$?) for the general case. With integer edge weights this can be reduced to linear time. While the Greedy Algorithm is rather simple it still is a 1/2-approximation (see Theorem 13).

Theorem 13
$w(\mathcal{M}_{Opt}) \leq 2 \cdot w(\mathcal{M}_{Greedy})$ *for non-negative* w.

Proof. Let $w(e)$ be the weight of the first selected edge. After removing e and its incident edges there are at most two edges of an optimal matching removed and the sum of their weights cannot exceed $2w(e)$. Repeating the argument for the remaining iterations yields the claim. □

Heavy Edge Matching. A drawback of the greedy algorithm is that we have to sort all edges. The HEM-Algorithm (see Algorithm 8) avoids this by visiting the nodes in a random order and taking the heaviest edge the current node under consideration. The algorithm can be implemented to run in $O(m)$ time. A simple implementation achieving this is the following: store an array of boolean values that encodes if a node is matchable. Initially, all values of the array are true. When we visit a node, we iterate over all adjacent edges and pic the heaviest edge that is still matchable. Once we found a matching edge, the incident vertices are marked as not matchable. Unfortunately, the algorithm is *not* an approximation algorithm. An construction illustrating this can be found in Figure 8.2.

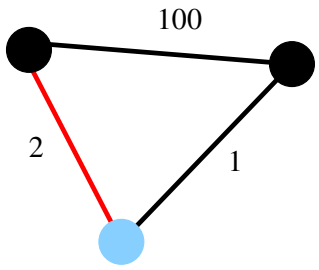


Figure 8.2: The weight of the maximum weight matching for this graph is 100, while the HEM Algorithm only returns a matching of weight 2, if we start with the bottom node (blue). By increasing the weight of the heaviest edge, we can get arbitrarily bad performance.

Algorithm 8 HEM Algorithm

$\mathcal{M}_{HEM} := \emptyset$
for $v \in V$ in random order **do**
  take *heaviest* edge $\{v, w\} \in E$
  add $\{v, w\}$ to $\mathcal{M}_{HEM}$
  remove all edges incident to v or w from E
end for

Global Paths Algorithm. The Global Paths Algorithm (GPA), was proposed by Maue et al. [92] as a synthesis of Greedy and Path Growing algorithms by Drake et al. [44]. The greedy algorithm sorts the edges by descending weight (or rating) and then scans them. If an edge $\{u, v\}$ and its end points are not matched yet, it is put into the matching.

Similar to the Greedy approach, GPA scans the edges in order of decreasing weight; but rather than immediately building a matching, it first constructs a collection of paths and even length cycles. To be more precise, these paths initially contain no edges. While scanning the edges, the set is extended by successively adding applicable edges. An edge is called applicable if it connects two endpoints of different paths or the two endpoints of an odd length path. Afterwards, optimal solutions/matchings are computed for each of these paths and cycles using dynamic programming (how?). Both algorithms achieve a half-approximation in the worst case, but empirically, GPA gives considerably better results [92]. The first algorithm that achieved a half-approximation was given by Preis [110]. In the context of graph partitioning it has been shown that the GPA algorithm is a good choice when focusing on partitioning quality [66]. The resulting algorithm can be found in Algorithm 9.

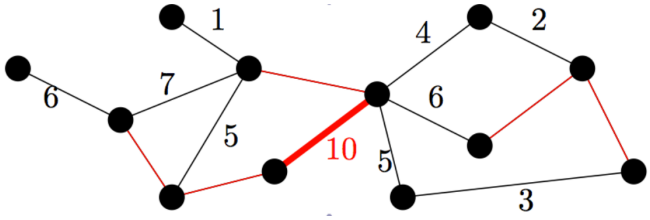


Figure 8.3: A figure illustrating the global paths algorithm.

Lemma 14
Let $\mathcal{M}_{Opt}$ be the optimal maximum weight matching and E' be the set of paths and cycle edges found by the GPA Algorithm. Then: $w(E') \geq w(\mathcal{M}_{Opt})$.

Proof. All edges $e \in \mathcal{M}_{Opt} \setminus E'$ were not applicable at runtime. Thus for all those edges exist at least two distinct edges e_1, e_2 with $w(e_1) > w(e)$ and $w(e_2) > w(e)$. We also know, that all edges in E' are adjacent to at most two edges in $\mathcal{M}_{Opt}$. Therefore we can construct an injective function $f : \mathcal{M}_{Opt} \to E', w(f(e)) \geq w(e)$. Since the GPA Algorithm finds the exact optimal maximum weighed matching for the set of paths and even-length cycles E', we know that $w(E') \geq w(\mathcal{M}_{Opt})$. □

Lemma 15
A maximum weight matching of a path or even-length cycle has at least half the weight of all edges on it.

Theorem 16
$w(\mathcal{M}_{GPA}) \geq \frac{1}{2} w(\mathcal{M}_{Opt})$ *for non-negative* w *[9].*

Proof. Using Lemma 14, we know that $\frac{1}{2} w(E') \geq \frac{1}{2} \mathcal{M}_{Opt}$. We also know that a maximum weight matching for E' has at least half the weight of E' (see Lemma 15).

Algorithm 9 GPA Algorithm

$\mathcal{M}_{GPA} := \emptyset$
$E' := \emptyset$
for all $e \in E$ in descending order **do**
  if e is *applicable* **then**
    add e to E'
  end if
end for
for all paths or cycles P in E' **do**
  $M' :=$ MaximumWeightMatching(P)
  $\mathcal{M}_{GPA} = \mathcal{M}_{GPA} \cup M'$
end for

Thus we know that:

$$w(\mathcal{M}_{GPA}) \geq \frac{1}{2} w(E') \geq \frac{1}{2} \mathcal{M}_{Opt} \tag{8.1}$$

Lemma 17
The ratio of the approximation in Lemma 14 is tight.

Proof. The graph in Figure 8.4 has a maximum weight matching of weight $w(\mathcal{M}_{Opt}) = \frac{m}{2} c$, while the GPA Algorithm finds a matching with a weight of at most $\frac{m}{4}(c + \epsilon) = \frac{1}{2} w(\mathcal{M}_{Opt}) + \epsilon'$.

Preis Principle. As shown above we cannot give a performance guarantee for the HEM Algorithm. This is unfortunate, since the algorithm runs in time $O(m)$ and is conceptually very simple. We can change the algorithm slightly to get an 1/2-approximation: instead of adding the heaviest edge for a node to the matching we only add edges which are locally maximal, i.e. no incident edge is heavier.

The performance of this algorithm is $O(m)$, since we still only need to visit each edge twice. To show why this algorithm is an 1/2-approximation we can use the same argument we used for the Greedy Algorithm.

Local Max Algorithm. While most of the algorithms discussed in this chapter work well, they still have one mayor downside: they are not trivial to scale. To overcome this we can use the Local Max Algorithm. Given a distributed system we can use Preis Principle to remove all locally maximal edges in parallel in each iteration. This approach is called the Local Max Algorithm.

At first glance this algorithm is neither fast nor efficient, since we need $\Omega(n)$ iterations to find a matching for the graph given in Figure 8.5. But the algorithm can be quite fast in practice for random edge weights (see Theorem 18).

Theorem 18
For random edge weights the Local Max Algorithm takes $O(\log m)$ iterations in expectation to find a matching and does linear work, i.e. $O(m)$, overall.

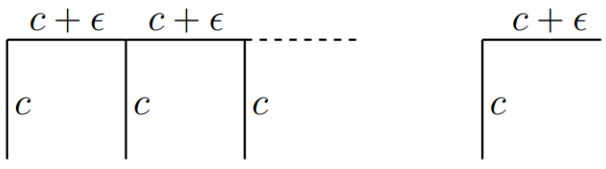


Figure 8.4: A graph were the GPA Algorithm yields a matching with a weight of at most $\frac{m}{4}(c + \epsilon)$.

Figure 8.5: A graph were the Local Max Algorithm takes at least n iterations to find a matching. Heavier edges are drawn thicker.

Proof. Due to Lemma 19 we know that in the expected case the number of edges is halved in each iteration. Thus we can expect to have a logarithmic number of iterations until no edges are left and the algorithm terminates. The overall work can be described as a geometric series for the expected case which results in a linear amount of overall work.

Lemma 19
For random edge weights the number of edges is halved in expected in each iteration of the Local Max Algorithm.

Proof. To prove Lemma 19 we introduce a mark for each end-point of and edge. During one iteration we mark the edge $\{u, v\}$ at the v-side, iff v is matched. We see that an edge $\{u', v'\}$ is removed in this iteration, iff it gets at least one mark. Thus matching an edge $\{u, v\}$ leads to $d(u) + d(v)$ marks, if $d(v)$ is the degree of the node v. We now introduce a random variable $\mathbf{X}_{\{u,v\}}$ which represents the number of marks introduced by the edge $\{u, v\}$. This value can only be 0 or $d(u) + d(v)$. We can compute the total number of marks as $X = \sum_{e \in E} \mathbf{X}_e$. Lemma 20 shows, that $E[X] \geq m$. Thus we know that at least $m/2$ edges have at least one mark. Due to this the expected value for the number of edges removed in one iteration is at least $m/2$, which proofs the lemma.

Lemma 20
$\mathbb{E}[X] \geq m$

Proof.

$$\begin{aligned}
\mathbb{E}[\mathbf{X}] &= \mathbb{E}[\sum_{e \in E} \mathbf{X}_e] \\
&= \sum_{e \in E} \mathbb{E}[\mathbf{X}_e] \\
&= \sum_{\{u,v\} \in E} (d(u) + d(v)) \mathbf{P}[\{u, v\} \text{ is locally maximal}] \\
&= \sum_{\{u,v\} \in E} (d(u) + d(v)) \frac{1}{d(u) + d(v) - 1} \\
&\geq \sum_{\{u,v\} \in E} \frac{d(u) + d(v)}{d(u) + d(v)} = \sum_{e \in E} 1 = m
\end{aligned}$$

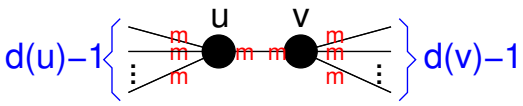


Figure 8.6: The number of edges adjacent to an edge $\{u, v\}$ is $d(u) + d(v) - 1$, if we also count the edge $\{u, v\}$.

To see why $\mathbf{P}[\{u, v\} \text{ is locally maximal}] = \frac{1}{d(u)+d(v)-1}$ holds, we use that exactly $d(u) + d(v) - 1$ edges are adjacent to an edge $\{u, v\}$, including the edge itself (see Figure 8.6). The chance that one of those has the greatest weight is equal for each edge due to the random edge weights in the graph. Thus the chance that the edge $\{u, v\}$ has the highest weight is $\frac{1}{d(u)+d(v)-1}$.

An implementation for the Local Max Algorithm can be found in Algorithm 10. This algorithm can for instance be implemented in a distributed memory model, like MPI. In this case all nodes are evenly distributed over all processors. A processor also receives all edges for each of its nodes. Thus edges can be present on multiple processors (see Figure 8.7). Thus the first for-loop of the algorithm can be performed locally on each processor. The second for-loop is performed as locally as possible, but if needed processors exchange candidate information. The last for-loop can again be computed locally. There are also implementations for this algorithm for vertex centric models like Apache Giraph or Pregel.

Algorithm 10 Local Max Algorithm

$\mathcal{M}_{LMA} := \emptyset$
while $E \neq \emptyset$ **do**
  for all $v \in V$ **do**
    $C[v] = \underset{\{u,v\} \in E}{\operatorname{argmax}}(w(\{u, v\}))$
  end for
  for all $\{u, v\} \in E$ **do**
    if $C[u] = C[v]$ **then**
      $\mathcal{M}_{LMA} = \mathcal{M}_{LMA} \cup \{\{u, v\}\}$
      mark u and v as matched
    end if
  end for
  for all $\{u, v\} \in E$ **do**
    if u or v is marked as matched **then**
      $E = E \setminus \{\{u, v\}\}$
    end if
  end for
end while

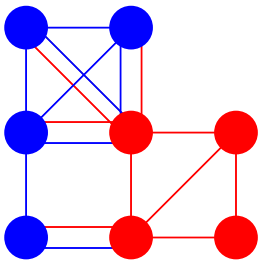


Figure 8.7: The distribution of nodes to two processors in an MPI-implementation of the Local Max Algorithm. Blue nodes and edges are assigned to one processor and red nodes to the other one. For all edges that are both blue and red candidate information needs to be exchanged in the second for-loop of the algorithm.

8.2.1 Experiments

Since most of the algorithms discussed in this chapter are 1/2-approximations we need experiments to see which algorithm performs best in practice. Figure 8.8 shows the weight of the matching returned by the Greedy Algorithm, the HEM Algorithm and the Red-Blue Matching Algorithm (which we did not discuss here). All weights are ratios compared to the Global Paths Algorithm. Thus this figure can be seen as an overall comparison of the quality of the different algorithms. As we can see the HEM Algorithm sometimes performs quite good, but there are many cases were the weight ratio is far below the other algorithms. On the other hand the Greedy Algorithm performs surprisingly well with an average weight ratio near 1. The Local Max Algorithm performs similarly.

A comparison of the running time of the algorithms can be found in Figure 8.9. For this comparison Delaunay instances were used as input graphs. Delaunay instances are random graphs created by randomly choosing n points in the united square and computing the Delaunay Triangulation to create the edges. Edge weights are assigned as euclidean distances scaled to integer space. As we can see the HEM Algorithm performs best, but the running time per edge grows with larger instances. The Performance of the Local Max Algorithm and the Greedy Algorithm are surprisingly similar to each other, especially for larger instances. The RBM Algorithm and the Global Path Algorithm are around a factor two slower than Greedy and Local Max.

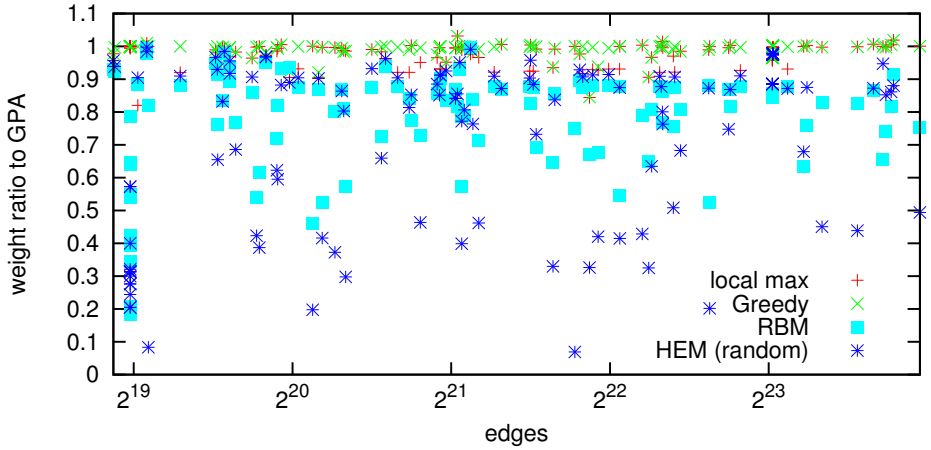


Figure 8.8: Weight ratio of the Red-Blue Matching Algorithm, the Greedy Algorithm, the Local Max Algorithm and the HEM Algorithm. The Global Path Algorithm was used as baseline.

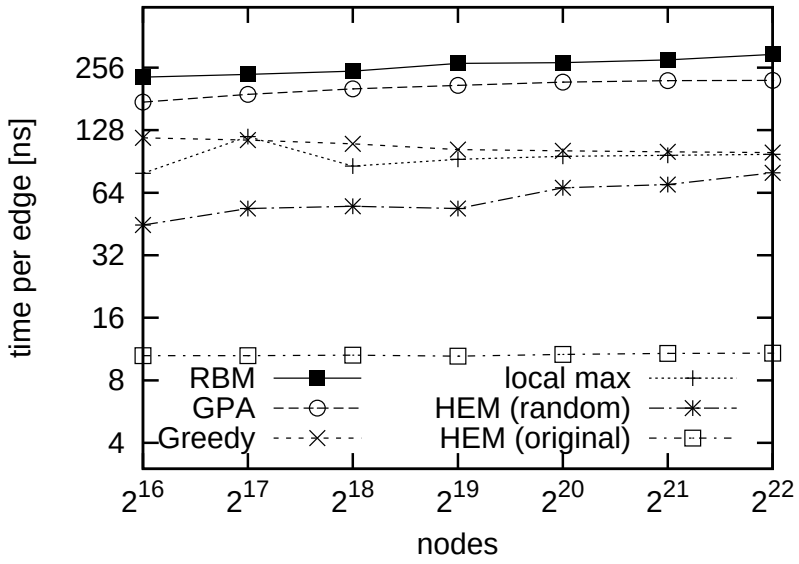


Figure 8.9: Running times per edge for the Red-Blue Matching Algorithm, the Global Path Algorithm, the Greedy Algorithm, the Local Max Algorithm and the HEM Algorithm.

8.2.2 Edge Ratings

To explain the concept of edge ratings, we have to characterize the properties of "good" matchings for the purpose of contraction in a multilevel algorithm for graph partitioning. Intuitively, a matching should contain *large edge weights* since we want to solve the problem on the coarsest level and our main objective is to find a small cut. On the other hand a matching should contain a *large number of edges*, e.g. being maximal, so that there are only few levels in the hierarchy and the algorithm can converge quickly. In order to *represent the input* on the coarser levels, we want to find matchings such that the graph after contraction has somewhat *uniform node weights* and *small node degrees*. Uniform node weights are also helpful to achieve a balanced partition on the coarser levels and makes local search algorithms more effective. Using the *edge weight* as a rating function has been done before and takes care of the first two requirements. However, the edge weight completely ignores the two latter properties of a good matching.

The perspective taken in KaPPa [66] is to encode these properties into a single scalar edge rating function. Then an approximate weight matching algorithm is applied that tries to find a matching which maximizes the sum of the ratings. Two good rating functions are:

$$\begin{aligned} \text{expansion}^{*2}(\{u,v\}) &:= \omega(\{u,v\})^2/c(u)c(v), \text{ and} \\ \text{innerOuter}(\{u,v\}) &:= \omega(\{u,v\})/(\text{Out}(v)+\text{Out}(u)-2\omega(u,v)), \end{aligned}$$

where $\text{Out}(v)$ is set to $\text{Out}(v) := \sum_{x\in\Gamma(v)} \omega(\{v,x\})$. For the purpose of partitioning unstructured networks, we also look at a rating based on algebraic distance as a measure of connectivity between the nodes.

The notion of *algebraic distance* has been introduced by Safro et al. [113, 33] and is based on the principle of obtaining low-residual error components [25]. This principle was used for linear ordering problems to distinguish between *local* and *global* edges [113]. A local edge e is characterized by having a small distance between its end nodes after e is removed. In contrast to a local edge, a non-local or global edge is defined via a large distance of its end nodes in the graph after e is removed. The main difference between the graph partitioning problem and linear ordering problems is the balance constraint. Thus, we introduce a node weight *normalized algebraic distance* to ensure that the node weights do not get too non-uniform during coarsening.

Given the Laplacian of a graph $L = D - W$, where W is a weighted adjacency matrix of a graph and D is the diagonal matrix with entries $D_{vv} = d(v)$, we define its node weight normalized version by $\tilde{L} = \tilde{D} - \tilde{W}$ based on the normalized edge weights $\tilde{\omega}(e = \{v,w\}) := \omega(e)/\sqrt{c(v)c(w)}$. We then define an iteration matrix $\tilde{H}$ for Jacobi over-relaxation as

$$\tilde{H} = (1-\alpha)I + \alpha\tilde{D}^{-1}\tilde{W},$$

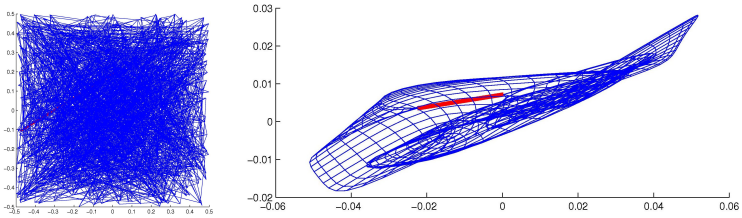


Figure 8.10: Left: a 12x40 grid with random coordinates. A non-local edge is highlighted in red. Right: Coordinates of nodes have been updated using 15 JOR iterations. The length of the red edge is larger than the length of local edges.

where $0 \leq \alpha \leq 1$. Note that the original matrix $H_{\text{JOR}} = (1-\alpha)I + \alpha D^{-1}W$ is basically the JOR-matrix to solve the system $Lx = 0$. For $\alpha = 1/2$ the matrix is also known as a lazy random-walk matrix. However, instead of solving the system (which has the trivial solutions $x = 0$ and $x = 1$) one uses a few iterations of a process used to solve the linear system

$$x^k = \tilde{H}x^{k-1} = \tilde{H}^k x^0,$$

where x^0 is a random vector sampled over $[-1/2, 1/2]$. Intuitively, the process *assimilates* the random values in a neighborhood of a node. This is repeated R times using different random start vectors. The algebraic distance is then defined by averaging over the result vectors. More precisely, let $x^{(k,r)}$ be the result of the process of the rth try. In [33] it is conjectured that if $|x_i^k - x_j^k| > |x_u^k - x_v^k|$, then the connectivity between u and v is larger than the connectivity between i and j. Hence, for an edge $e \in E$ the 2-normed algebraic distance coupling ρ_e is then defined as

$$\rho_{\{u,v\}} = \sqrt{\sum_{r=1}^{R} |x_u^{(k,r)} - x_v^{(k,r)}|^2}.$$

If $e \notin E$, then the algebraic distance coupling is set to zero. In our experimental settings we use $\alpha = 0.5$, $R = 5$, and $k = 20$. Figure 8.10 gives an intuition on why ρ can be used as a measure of connectivity strength. Recall that if ρ_e is small then its end nodes are strongly coupled and otherwise loosely coupled. Having this in mind, we define an advanced edge rating function

$$\text{ex\_alg}(e) := \text{expansion}^{*2}(e)/\rho_e.$$

This rating function prefers edges for contraction that are strongly coupled with respect to algebraic distance. Note that the iterative process, usually a sparse matrix vector multiplication, can be parallelized easily.

8.3 Cluster-based Contraction

We now explain the basic idea of an approach to create graph hierarchies, which is targeted at complex network such as social networks and web graphs. We start by introducing the size-constrained label propagation algorithm, which is used to compute clusterings of the graph. To compute a graph hierarchy, the clustering is contracted by replacing each cluster with a single node, and the process is repeated recursively until the graph is small. Due to the way the contraction is defined, it is ensured that a partition of a coarse graph corresponds to a partition of the input network with the same objective and balance. Note that cluster contraction is an aggressive coarsening strategy. In contrast to previous approaches, it enables us to drastically shrink the size of irregular networks. The intuition behind this technique is that a clustering of the graph (one hopes) contains many edges running inside the clusters and only a few edges running between clusters, which is favorable for the edge cut objective.

8.3.1 Label Propagation with Size-Constraints

The *label propagation algorithm* (LPA) was proposed by Raghavan et al. [111] for graph clustering. It is a fast, near-linear time algorithm that locally optimizes the number of edges cut. We outline the algorithm briefly. Initially, each node is in its own cluster/block, i.e. the initial block ID of a node is set to its node ID. The algorithm then works in rounds. In each round, the nodes of the graph are traversed in a random order. When a node v is visited, it is *moved* to the block that has the strongest connection to v, i.e. it is moved to the cluster V_i that maximizes $\omega(\{(v,u) \mid u \in N(v) \cap V_i\})$. Ties are broken randomly. The process is repeated until the process has converged. Here, we perform at most ℓ iterations of the algorithm, where ℓ is a tuning parameter, and stop the algorithm if less then five percent of the nodes changed its cluster during one round. One LPA round can be implemented to run in $\mathrm{O}(n+m)$ time.

After we have computed a clustering, we *contract it* to obtain a coarser graph.

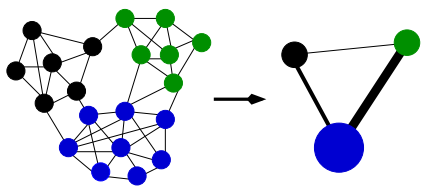


Figure 8.11: Contraction of a clustering. The clustering of the graph on the left hand side is indicated by the colors. Each cluster of the graph on the left hand side corresponds to a node in the graph on the right hand side.

Contracting the clustering works as follows: each block of the clustering is contracted into a single node. The weight of the node is set to the sum of the weight of all nodes in the original block. There is an edge between two nodes u and v in the contracted graph if the two corresponding blocks in the clustering are adjacent to each other in G, i.e. block u and block v are connected by at least one edge. The weight of an edge (A, B) is set to the sum of the weight of edges that run between block A and block B of the clustering. Note that due to the way the contraction is defined, a partition of the coarse graph corresponds to a partition of the finer graph with the same cut and balance. An example is shown in Figure 8.11.

In contrast to the original LPA [111], we have to ensure that each block of the cluster fulfills a size constraint. There are two reason for this. First, consider a clustering of the graph in which the weight of a block would exceed $(1 + \epsilon)\lceil \frac{|V|}{k} \rceil$. After contracting this clustering, it would be impossible to find a partition of the contracted graph that fulfills the balance constraint. Secondly, it has been shown that using more balanced graph hierarchies is beneficial when computing high quality graph partitions [66]. To ensure that blocks of the clustering do not become too large, we introduce an upper bound $U := \max(\max_v c(v), W)$ for the size of the blocks. Here, W is a parameter that will be chosen later. When the algorithm starts to compute a graph clustering on the input graph, the constraint is fulfilled since each of the blocks contains exactly one node. A neighboring block V_ℓ of a node v is called *eligible* if V_ℓ will not become overloaded once v is moved to V_ℓ. Now when we visit a node v, we move it to the *eligible block* that has the strongest connection to v. Hence, after moving a node, the size of each block is still smaller than or equal to U. Moreover, after contracting the clustering, the weight of each node is smaller or equal to U. One round of the modified version of the algorithm can still run in linear time by using an array of size $|V|$ to store the block sizes. We set the parameter W to $\frac{L_{\max}}{f \cdot k}$, where f is a tuning parameter.

By using a different size-constraint – the constraint $W := L_{\max}$ of the original partitioning problem – the LPA can also be used as a simple and fast local search algorithm to improve a solution on the current level. However, one has to make small modifications to handle overloaded blocks. We modify the block selection rule when we use the algorithm as local search algorithm in case that the current node v under consideration is from an overloaded block V_ℓ. In this case it is *moved* to the eligible block that has the strongest connection to v without considering the block V_ℓ it is contained in. This way it is ensured that the move improves the balance of the partition (at the cost of the number of edges cut).

8.3.2 Algorithmic Extensions

In this section we present numerous algorithmic extensions to the approach presented above. This includes using different orderings for size-constrained LPA,

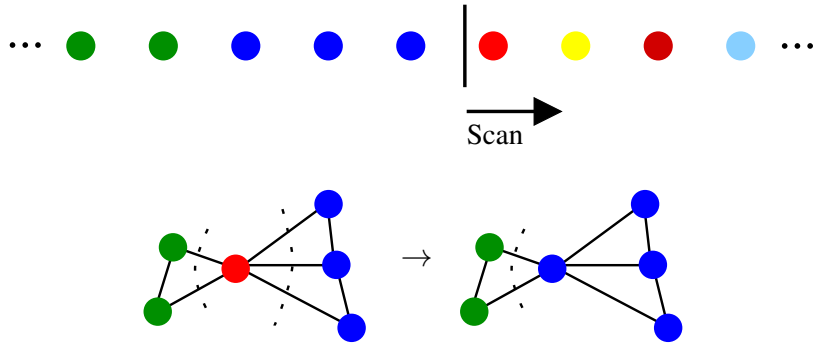


Figure 8.12: Example run of label propagation.

combining multiple clusterings into one clustering, advanced multilevel schemes, allowing additional amounts of imbalance on coarse levels of the multilevel hierarchy and a method to improve the speed of the algorithm.

Node Ordering for Label Propagation. The LPA traverses the nodes in a random order and moves a node to a cluster with the strongest connection in its neighborhood to compute a clustering. Instead of using a random order, one can use the ordering induced by the node degree (increasing). That means that in the first round of the label propagation algorithm, nodes with small node degree can change their cluster before nodes with a large node degree. Intuitively, this ensures that there is already a meaningful cluster structure when the LPA chooses the cluster of a high degree node. Hence, the algorithm is likely to compute better clusterings of the graph by using node orderings based on node degree. We also tried other node orderings such as weighted node degree. The overall solution quality and running time are comparable so that we omit more sophisticated orderings.

Ensemble Clusterings. In machine learning, ensemble methods combine multiple weak classification (or clustering) algorithms to obtain a strong algorithm for classification (or clustering). Such an ensemble approach has been successfully applied to graph clustering by combining several base clusterings from different LPA runs. These base clusterings are used to decide whether pairs of nodes should belong to the same cluster [131]. We follow the idea to get *better* clusterings for the coarsening phase of our multilevel algorithm.

Given a number of clusterings, the *overlay clustering* is a clustering in which two nodes belong to the same cluster if and only if they belong to the same cluster in each of the input clusterings. Intuitively, if all of the input clusters agree that two nodes belong to the same block, then they are put into the same block in the overlay clustering. On the other hand, if there is one input clustering that puts the nodes

into different blocks, then they are put into different blocks in the overlay clustering. More formally, given clusterings $\{\mathcal{C}_1, \ldots, \mathcal{C}_\ell\}$, we define the overlay clustering as the clustering where each block corresponds to a connected component of the graph $G_\mathcal{E} = (V, E \backslash \mathcal{E})$, where $\mathcal{E}$ is the union of the cut edges of each of the clusterings $\mathcal{C}_i$, i.e. all edges that run between blocks in the clusterings $\mathcal{C}_i$. Related definitions are possible, e.g. a cluster does not have to be a connected component. In our ensemble approach we use the clusterings obtained by size-constrained LPA as input to compute the overlay clustering. It is easy to see that the number of clusters in the overlay clustering cannot decrease compared to the number of clusters in each of the input clusterings. Moreover, the overlay clustering is feasible w.r.t. to the size constraint if each of the input clusterings is feasible.

Given ℓ clusterings $\{\mathcal{C}_1, \ldots, \mathcal{C}_\ell\}$, we use the following approach to *compute* the overlay clustering iteratively. Initially, the overlay clustering O is set to the clustering $\mathcal{C}_1$. We then iterate through the remaining clusterings and incrementally update the current solution O. This is done by computing the overlay $\mathcal{O}$ with the current clustering $\mathcal{C}$ under consideration. More precisely, we use pairs of cluster IDs (i, j) as a key in a hash map $\mathcal{H}$, where i is a cluster ID of $\mathcal{O}$ and j is a cluster ID of the current clustering $\mathcal{C}$. We then iterate through the nodes and initialize a counter c to zero. Let v be the current node. If the pair $(\mathcal{O}[v], \mathcal{C}[v])$ is not contained in $\mathcal{H}$, we set $\mathcal{H}(\mathcal{O}[v], \mathcal{C}[v])$ to c and increment c by one. Afterwards, we update the cluster ID of v in $\mathcal{O}$ to $\mathcal{H}(\mathcal{O}[v], \mathcal{C}[v])$. Note that at the end of the algorithm, c is equal to the number of clusters contained in the overlay clustering. Moreover, it is possible to compute the overlay clustering directly by hashing ℓ-tuples [131]. However, we choose the simpler approach here since the computation of a clustering itself already takes near-linear time.

Iterated Multilevel Algorithms. As mentioned earlier, a common approach to obtain high quality partitions is to use a multilevel algorithm multiple times using different random seeds and use the best partition that has been found. However, one can do better by transferring the solution of the previous multilevel iteration down the hierarchy. For simplicity, we focus on iterated V-cycles which are illustrated in Figure 5.3. We *adopt this technique* also for our new coarsening scheme by ensuring that cut edges are not contracted after the first multilevel iteration.

We have to ensure that cut edges are not contracted after the first multilevel iteration. We do this by modifying the label propagation algorithm such that each cluster of the computed clustering is a subset of a block of the input partition. In other words, each cluster only contains nodes of one unique block of the input partition. Hence, when contracting the clustering, every cut edge of the input partition will remain. Recall that the label propagation algorithm initially puts each node in its own block so that in the beginning of the algorithm each cluster is a subset of one unique block of the input partition. To keep this property during the

course of the label propagation algorithm, we restrict the movements of the label propagation algorithm, i.e. move a node to an eligible cluster with the strongest connection in its neighborhood that is in the same block of the input partition as the node itself. More precisely, let $V = U_1 \cup \ldots \cup U_k$ be the partition of the graph in the current level of the multilevel hierarchy. When a node $v \in U_\ell$ is visited, it is moved to an eligible cluster $V_i \subseteq U_\ell$ that maximizes $\omega(\{(v,u) \mid u \in N(v) \cap V_i\})$.

Allowing Larger Imbalances on Coarse Levels. It is well-known that temporarily allowing larger imbalance is useful to create good partitions [135, 119]. Allowing an additional amount of imbalance $\hat{\epsilon}$ means that the balance constraint is relaxed to $(1 + \epsilon + \hat{\epsilon})\lceil \frac{|V|}{k} \rceil$, where ϵ is the original imbalance parameter and $\hat{\epsilon}$ is a parameter that has to be set appropriately. We adopt a simplified approach in this context and decrease the amount of additional allowed imbalance level-wise. In other words the largest amount of additional imbalance is allowed on the coarsest level and it is decreased level-wise until no additional amount of imbalance is allowed on the finest level. To be more precise, let the levels of the hierarchy be numbered in increasing order $G_1, \ldots, G_q$ where G_1 is the input graph G and G_q is the coarsest graph. The amount of allowed imbalance on a coarse level ℓ is set to $\hat{\epsilon}_\ell = \delta/(q - \ell + 1)$, where δ is a tuning parameter. No additional amount of imbalance is allowed on the finest level. Moreover, we only allow a larger amount of imbalance during the first V-cycle.

Active Nodes. The LPA looks at every node in each round of the algorithm. Assume for now that LPA is run without a size-constraint. After the first round of the algorithm, a node can only change its cluster if one or more of its neighbors changed its cluster in the previous round (for the sake of the argument we assume that ties are broken in exactly the same way as in the previous round). The active nodes approach keeps track of nodes that can potentially change their cluster. A node is called *active* if at least one of its neighbors changed its cluster in the *previous round*. In the first round all nodes are active. The original LPA is then modified so that only active nodes are considered for movement. This algorithm is always used when the label propagation algorithm is used as a local search algorithm during uncoarsening. A round of the modified algorithm can be implemented with running time linear in the amount of edges incident to the number of active nodes.

8.4 AMG-inspired Coarsening

When one wants to solve a system of linear equations, one of the most traditional approaches to creating hierarchies in Algebraic Multigrid (AMG) is the Galerkin operator [132], which projects a fine system of equations to a system of coarser scale. In the context of graphs this projection is defined as

$$L_c = PL_fP^T,$$

where L_f and L_c are the Laplacians of fine and coarse graphs $G_f = (V_f, E_f)$ and $G_c = (V_c, E_c)$, respectively and P is the projection matrix. The (u, J)th entry of the projection matrix P represents the strength of the connection between a fine node u and a coarse node J. The entries of P are called interpolation weights. They describe both the coarse-to-fine and fine-to-coarse relations between nodes.

The coarsening begins by selecting a dominating set of coarse (or seed) nodes $C \subset V_f$ such that all other fine nodes in $F = V_f \setminus C$ are strongly coupled to C. These nodes will serve as nodes of the next coarser level. The remaining nodes will be split among those nodes using the interpolation matrix P. The whole process is illustrated in Figure 8.13. The *selection* of the seed nodes can be done by traversing all nodes, moving nodes from F to C until the following equation is satisfied for all nodes u in F (initially $F = V_f$, and $C = \emptyset$)

$$\sum_{v \in C \cap N(u)} 1/\rho_{uv} \geq \Theta \cdot \sum_{v \in N(u)} 1/\rho_{uv},$$

where $\Theta \in (0, 1)$ is a parameter of coupling strength which is usually set to 0.5. Note that the notion of algebraic distance is used in these equations and also later on when we compute the coupling strength. Note that initially the constraint is not fulfilled. Moreover, C is a dominating set if the constraint is fulfilled and the graph does not contain singletons. To see this assume that there is a node in F that is only adjacent to nodes in the same set. In this case, the left hand side of the equation is zero which means that the constraint cannot be fulfilled.

We now have found the seed nodes for the next coarser level and start to explain how the *projection matrix* P is *constructed*. The projection matrix controls how fractions of a node from F are assigned to the seed nodes of the coarser level. The construction differs from other AMG-based approaches for combinatorial optimization problems since the graph partitioning problem demands *balanced* blocks and graphs should stay sparse on coarse levels of the hierarchy. To achieve this, we avoid too heavy coarse nodes and limit the number of fractions that a node from F can be divided to. In our algorithm the number of fractions is limited to at most two (the two strongest connections).

More precisely, the construction works as follows. The entries P_{vv} of seed nodes v are set to one. For a node $v \in F$ that is not a seed node for the coarse level,

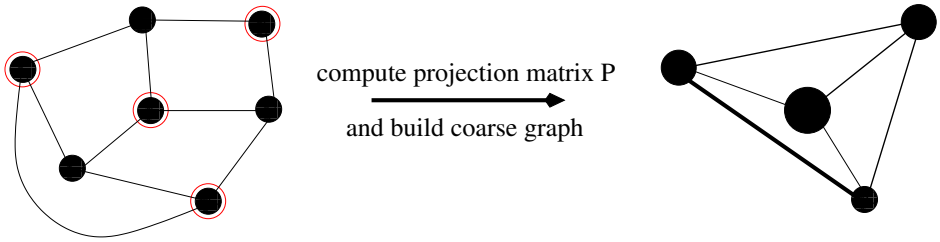


Figure 8.13: (a) Left: begin by selecting a dominating set C (circled nodes), these nodes will serve as nodes of the coarse graph. Compute a projection matrix to estimate the weight of the edges connecting the coarse nodes and to split fine nodes between incident seed nodes. Right: the resulting coarse graph using the dominating set as coarse nodes. The weights of the nodes are also updated.

we try to find two neighboring seed nodes that maximize the connection strength $(1/\rho_{e_1} + 1/\rho_{e_2})$ to v and that do not become overloaded when v is split among them. Here, we only look at the κ strongest connections. If we do not succeed to find such a pair, we find the strongest connection, w.r.t $1/\rho_e$, to a neighboring seed node such that it does not become overloaded if v is assigned to this seed node. It might still happen that we do not find such a node. In this case, we make v a seed node, i.e. we put it into C. The different stages of the algorithm are sketched in Figure 8.14. Having found the nodes $S \subset C$ that v is split among, we set the entry in the projection matrix P_{vc} depending on the connection strength to $\frac{1/\rho_{vc}}{\sum_{k \in S} 1/\rho_{vk}}$ for $c \in S$. The weight of a coarse node is set to $c(q \in C) := \sum_v c(v) P_{v,q}$ and the weight of an edge between two coarse nodes is (according to the projection equation) $\omega(c_1, c_2) := \sum_{u \neq v} P_{u,c_1} \omega(u, v) P_{v,c_2}$.

The algorithm can be viewed as a simplified version of the algorithm presented in [25] with the additional restriction on the size of coarse nodes and the possibility to adaptively control the number of fractions that a node can be split in. P_{uq} thus represents the likelihood of i belonging to the qth aggregate. We emphasize the adaptivity of the set C, which is updated if seed nodes would become too heavy.

Also, recall that once a partition of a coarse graph is transferred to a partition of the next finer graph in the multilevel hierarchy, local improvement methods try to reduce the cut size. In the matching case the projection is easily done by assigning the node to the block of its coarse representative. For the AMG-inspired coarsening scheme the projection is more sophisticated. In particular, we have to choose a block to which a fine node is assigned to. This is due to the fact that fractions of it can be assigned to different coarse nodes and hence to different blocks. We assign a fine node v to the block that minimizes $\text{cut}_B \cdot p_B(v)$, where cut_B is the cut after v would be assigned to block B and $p_B(v)$ is a penalty function to avoid blocks that are heavily overloaded. To be more precise, after some experiments we fixed the

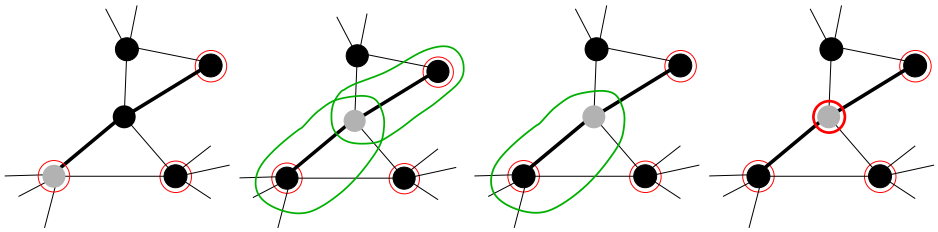


Figure 8.14: Different cases for the construction of interpolation weights P. Seed nodes are surrounded by a circle. (a) For seed nodes v, we set $P_{vv} = 1$. (b) The algorithm first tries to split a node between two seed nodes such that the seed nodes will not be overloaded when the grey node is split among them. (c) If this is not successful, the algorithm tries to find a seed node that is not overloaded when the grey node is aggregated with it. (d) If (b) and (c) are not successful, then the grey node is added to the set of seed nodes. Source: [114].

penalty function to

$$p_B(v) = 2^{\max(0,100\frac{c(B)+c(v)}{L_{\max}})},$$

where $L_{\max}$ is the upper bound for the block weight. Note that slight imbalances (e.g. overloaded blocks), can usually be fixed by our local search algorithms. Also observe that when using the AMG-inspired coarsening scheme, the cuts and balance of partition of a coarse level are *not* the same as the cut and balance of the partition on the next finer level after projection. This is in contrast to the matching based scheme.

We now discuss the differences between our scheme and the weighted aggregation (WAG) scheme by Chevalier et al. [35]. Both schemes assign fine F-nodes to their coarse C-neighbors. However, algebraic distance is missing which is responsible for the seed sets and the number of fractions a fine node is split into. The projection matrix P is constructed as in classical AMG schemes. Moreover, the WAG scheme produces dense coarse graphs, which we avoid by splitting a node into at most two coarse nodes.

8.5 n-level Graph Partitioning

The central idea of n-level graph partitioning by Osipov and Sanders [102] is to make subsequent levels as similar as possible – (un)contract only a *single* edge between two levels. n-level graph partitioning has the additional advantage that there is no longer a need for an algorithm finding heavy matchings. This is remarkable insofar as a considerable amount of work on approximate maximum weight matching was motivated by the MGP application (e.g. [110, 92]). Still, at first glance, n-GP seems to have substantial disadvantages also. Firstly, storing each level explicitly would lead to quadratic space consumption. We avoid this by using a dynamic graph data structure with little space overhead. Secondly, choosing maximal matchings instead of just a single edge for contraction has the side effect that the graph is contracted everywhere, leading to a more uniform distribution of node weights. We solve this problem by explicitly factoring node weights into the *edge rating* function prioritizing the edges to be contracted. Already in [66] edge ratings have proven to lead to better results for graph partitioning. Perhaps the most serious problem is that the most common approach to local search is to let it run for a number of steps proportional to the current number of nodes. In the context of n-GP this could lead to a quadratic overall number of local search steps. Therefore, we develop a new, more adaptive stopping criteria for the local search that drastically accelerates n-GP without significantly reducing partitioning quality. Algorithm 11 gives a high-level recursive summary of n-GP. The edges to be contracted are chosen according to an edge rating function. KaSPar adopts the rating function

$$\text{expansion}^*(\{u,v\}) := \frac{\omega(\{u,v\})}{c(u)c(v)}$$

which fared best in [66]. As a further measure to avoid unbalanced inputs to the initial partitioner, KaSPar never allows a node v to participate in a contraction if the weight of v exceeds $1.5n/(20k)$. Selecting contracted edges can be implemented efficiently by keeping the contractable *nodes* in a priority queue sorted by the rating of their most highly rated incident edge.

In order to make contraction and uncontraction efficient, we use a "semidynamic" graph data structure: When contracting an edge $\{u,v\}$, we mark both u

Algorithm 11 n-GP(G,k,ϵ)

if G is small **then return** initialPartition
pick the edge $e = \{u,v\}$ with the highest rating
contract e; $\mathcal{P} := n$-GP(G,k,ϵ); uncontract e
activate(u); activate(v); localSearch(G)

and v as deleted, introduce a new node w, and redirect the edges incident to u and v to w. The advantage of this implementation is that edges adjacent to a node are still stored in adjacency arrays which are more efficient than linked lists needed for a full fledged dynamic graph data structure. A disadvantages of our approach is a certain space overhead. However, it is relatively easy to show that this space overhead is bounded by a logarithmic factor even if we contract edges in some random fashion. Experiments demonstrate experimentally that the overhead is actually often a small constant factor [102]. Indeed, this is not very surprising since the edge rating function is not random, but designed to keep the contracted graph sparse. Overall, with respect to asymptotic memory overhead, n-GP is no worse than methods with a logarithmic number of levels.

Local Search Strategy. The local search strategy is similar to the FM algorithm [46] that is also used in many other MGP systems. We now outline our variant and then discuss differences. Originally, all nodes are unmarked. Only unmarked nodes are allowed to be activated or moved from one block to another. Activating a node $v \in B'$ means that for blocks $\{B \neq B' : \exists \{v, u\} \in E \wedge u \in B\}$ we compute the gain of moving v to block B. Node v is then inserted into the priority queue P_B using $g_B(v)$ as the priority. We call a queue P_B eligible if the highest gain node in P_b can be moved to block B without violating the balance constraint for block B. Local search repeatedly looks for the highest gain node v in any eligible priority queue P_B and moves v to block B. When this happens, node v becomes nonactive and marked, the unmarked neighbors of v get activated and the gains of the active neighbors are updated. The local search is stopped if either no eligible nonempty queues remain, or one of the stopping criteria described below applies. After the local search stops, it is rolled back to the lowest cut state reached during the search (which is the starting state if no improvement was found). Subsequently all previously marked nodes are unmarked. Local search is repeated until no improvement is found.

The main difference to the usual FM-algorithm is that our routine performs a highly localized search starting just at the uncontracted edge. Indeed, our local search does nothing if none of the uncontracted nodes is a *border node*, i.e., has a neighbor in another block. Other FM-algorithms initialize the search with all border nodes. In n-GP the local search may find an improvement quickly after moving a small number of nodes. However, in order to exploit this case, we need a way to stop the search much earlier than previous algorithms which limit the number of steps to a constant fraction of the current number of nodes $|V|$.

Stopping Using a Random Walk Model. It makes sense to make a stopping rule more adaptive by making it dependent on the past history of the search, e.g.,

on the difference between the current cut and the best cut achieved before.

We model the gain values in each step as identically distributed, independent random variables whose expectation μ and Variance σ^2 is obtained from the previously observed p steps. In [102] it is shown how from these (purely heuristical, i.e., technically unwarranted) assumptions we can derive that it is unlikely that the local search will produce a better cut if

$$p\mu^2 > \alpha\sigma^2 + \beta \tag{8.2}$$

where α and β are tuning parameters and μ is the average gain since the last improvement. For the variance σ^2, we can also use the variance observed throughout the current local search. Parameter β is a base value that avoids stopping just after a small constant number of steps that happen to have small variance. Currently we set it to $\ln n$.

References

This section is largely based on Schulz [122], Maue and Sanders [92], Holtgrewe, Sanders and Schulz [66], Vitaly and Sanders [102], Walshaw [134], Safro, Sanders and Schulz [114] and Avis [9]. Results from Preis [110], Gabow [51] and Hoepman [65] were also used. This section was created by Sebastian Bayer, Christoph Hess, Marvin Teichmann and Christian Schulz.

Chapter 9

Parallel Graph Partitioning

In this chapter we look at parallel algorithms to solve the graph partitioning problem. Parallel graph partitioning algorithms are desirable for many reason. For example, for large simulations or even in large companies such as Facebook or Google, graphs are often too large to fit in the memory of a single processor. Therefore, a parallel graph partitioning algorithm can take advantage of the *large amount of memory* usable in parallel computers to solve even larger problems. On the other hand, in some scientific applications graph partitioning can become the *bottleneck* of computation and if graph partitioning is employed for the purpose of parallelization one wants to use the processors that are already available. For example, in adaptive finite element simulations the mesh is getting finer and finer at specific parts of the geometry using parallel mesh generation algorithms. The cost of communicating the mesh to a single processor for partitioning can be high and one would be waste of time to wait for the algorithm that solves the partitioning problem on a single machine.

The rest of the chapter is organized as follows. We start with a very simple and fast but low quality scheme, Recursive Coordinate Bissection. We continue with the description of the algorithms within ParMetis which is probably the most widely used parallel graph partitioner today. After looking into the details of the Karlsruhe Parallel Partitioner, we see to algorithms specialized for social networks – both of them based on parallel label propagation.

9.1 Recursive Coordinate Bisection

Recursive coordinate bisection (RCB) can be applied to compute a partition of the graph if the vertices of the graph have coordinates. A simple bisection strategy is then to determine the coordinate direction of longest expansion of the graph. Afterwards all vertices are sorted according to the respective coordinate direction

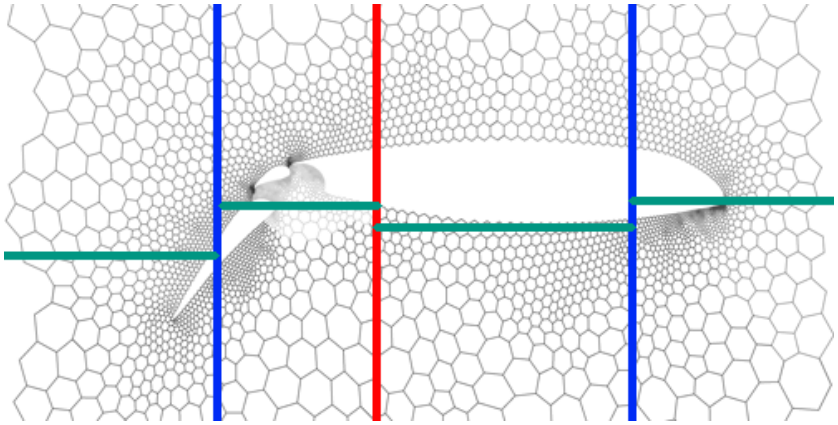


Figure 9.1: RCB of an airfoil

(one component of the coordinate vector). Half of the vertices with the smallest values in this direction are assigned to one block, the other half is assigned to the second block. The process is then repeated on each of the blocks until the total number of desired blocks is reached. The algorithm is easily parallelized by using parallel sorting algorithms or parallel selection algorithms to find the median of the coordinate direction. A brief outline of the algorithm can be found in Algorithm 12.

Algorithm 12 Recursive Bisection

```
Determine longest expansion of domain (x-, y-, or z-direction)
Sort vertices according to coordinate in selected direction
Assign half of the vertices to each block
Repeat recursively (divide and conquer)
```

There are pros and cons to this approach. For example, in practice coordinates are not always available and there is no control of the communication cost resulting from the computed partition, since the edge cut or the communication volume is at no point part of the objective of the algorithm. On the other hand, the algorithm is simple, easy to implement and fast and similar to $k-d$-trees.

9.2 ParMetis

ParMetis is a parallel version of the multilevel approach. Let p be the number of processors used to compute a p-way partitioning of the graph $G = (V, E)$. G is initially distributed among the processors using a one-dimensional distribution, so that each processor receives n/p vertices and their adjacency lists. At the end of the algorithm, a partition number is assigned to each vertex of the graph. In

the following sections we first describe the algorithm for computing a coloring of a graph on a distributed memory computer, and then describe our parallel formulations for the three phases of the multilevel k-way partitioning

Parallel Graph Coloring

A coloring of a graph $G = (V, E)$ assigns colors to the vertices of G so that adjacent vertices have different colors. We like to find a coloring such that the number of distinct colors used is small. The parallel graph coloring algorithm presented in this section consists of a number of iterations. In each iteration, a maximal independent set of vertices I is selected using a variation of Luby's algorithm. All vertices in this independent set are assigned the same color. Before the next iteration begins, the vertices in I are removed from the graph, and this smaller graph becomes the input graph for the next iteration. A maximal independent set I of a set of vertices S is computed in an incremental fashion using Luby's algorithm as follows. A random number is assigned to each vertex, and if a vertex has a random number that is smaller than all the random numbers of the adjacent vertices, it is then included in I. Now this process is repeated for the vertices in S that are neither in I nor adjacent to vertices in I, and I is augmented similarly. This incremental augmentation of I ends when no more vertices can be inserted in I. It is shown in that one iteration of Luby's algorithm requires a total of $O(\log |S|)$ such augmentation steps to find a maximal independent set of a set S.

Luby's algorithm can be implemented quite efficiently on a shared memory parallel computer, since for each vertex v, a processor can easily determine if the random value assigned to v is the smallest among all the random values assigned to the adjacent vertices. However, on a distributed memory parallel computer, for each vertex, random values associated with adjacent vertices that are not stored on the same processor need to be explicitly communicated. Furthermore, a faithful implementation of Luby's algorithm will also suffer from high synchronization overheads, as it requires a global synchronization step during each iteration. In the implementation of Luby's algorithm, we perform only a single augmentation step to compute the independent set during each iteration. Hence, the independent set computed is not maximal. Even though this leads to an increase in the number of colors required to color the entire graph, it significantly reduces the overall run time. Furthermore, we do not color all nodes of the graph, but stop when a large fraction of the graph is colored. This is acceptable because it still allows most of the nodes at any level to participate in the coarsening and local search phase while significantly limiting the required number of synchronization steps. In our implementation of Luby's algorithm, prior to performing the coloring in parallel, we perform a communication setup phase, in which appropriate data structures are created to facilitate this exchange of random numbers. In particular,

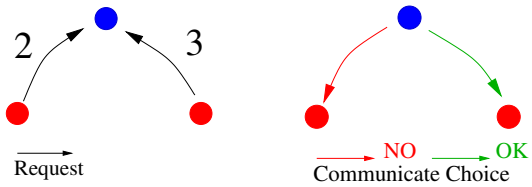
we predetermine which vertices are located on a processor boundary (i.e., a vertex connected with vertices residing on different processors) and which are internal vertices (i.e., vertices that are connected only to vertices on the same processors). These data structures are used in all the phases of our parallel multilevel graph partitioning algorithm.

Parallel Coarsening

The parallel matching algorithm is based on an extension of the serial heavy edge matching algorithm and utilizes graph coloring to structure the sequence of computations. Consider the current graph in the multilevel hierarchy G that has been colored using our parallel formulation of Luby's algorithm, and let Match be a variable associated with each vertex of the graph, which is initially set to -1. At the end of the computation, the variable Match for each vertex v stores the vertex to which v is matched. If v is not matched, then Match = v. To simplify the presentation, we first describe the algorithm assuming that the target parallel computer has a shared memory architecture, and later show how this algorithm is implemented on a distributed memory machine.

The matching is constructed in an iterative fashion. During the cth iteration, vertices of color c that have not yet been matched (i.e., Match = -1) select one of their unmatched neighbors using the heavy-edge heuristic, and modify the Match variable of the selected vertex by setting it to their vertex number. Let v be a vertex of color c and (v, u) be the edge that is selected by v. Since the color of u is not c, this vertex will not select a partner vertex at this iteration. However, there is a possibility that another vertex w of color c may select (w, u). Since both vertices v and w perform their selections at the same time, there is no way of preventing this. This is handled as follows. After all vertices of color c select an unmatched neighbor, they synchronize. The vertices of color c that have just selected a neighbor read the Match variable of their selected vertex. If the value read is equal to their vertex number, then their matching was successful, and they set their Match variable equal to the selected vertex; otherwise the matching fails, and the vertex remains unmatched. Note that if more than one vertex (e.g., v and w) want to match with the same vertex (e.g., u), only one of the writes in the Match variable of the selected vertex will succeed, and this determines which matching survives. However, by using coloring we restrict which vertices select partner vertices during each iteration; thus, the number of such conflicts is significantly reduced. Also note that, even though a vertex of color c may fail to have its matching accepted due to conflicts, this vertex can still be matched during a subsequent iteration corresponding to a different color.

The above algorithm is implemented quite easily on a distributed memory parallel computer as follows. The writes into the Match variables are gathered



(a) Request (b) Choice

Figure 9.2: Conflicts during parallel coarsening and how they are resolved

together and are sent to the corresponding processors in a single message. If a processor receives multiple write requests for the same vertex, the one that corresponds to the heavier edge is selected. Any ties are broken arbitrarily. Similarly, the reads from the Match variables are gathered by the processors that store the corresponding variables and they are sent in a single message to the requesting processors. Furthermore, during this read operation, the processors who own the Match variables also determine if they will be the ones storing the collapsed vertex in the coarser graph. This is done by using a uniformly distributed random variable. The vertex is kept or given away with the same probability. Our experiments have shown that this simple heuristic leads to a very good load balance.

After a matching is computed, each processor knows how many vertices (and the associated adjacency lists) it needs to send and how many it needs to receive. Each processor then sends and receives these subgraphs, and it forms the next-level coarser graph by merging the adjacency lists of the matched vertices. The coarsening process ends when the graph has $O(p)$ vertices.

Parallel Initial Partitioning

During the initial partitioning phase, a p-way partitioning of the graph is computed using a recursive bisection algorithm. Since the coarsest graph has only $O(p)$ vertices, this step can be performed serially without significantly affecting the performance of the entire algorithm. Nevertheless, in our algorithm we also parallelize this phase using a recursive decomposition. This is done as follows: the various pieces of the coarse graph are gathered to all the processors using an all-to-all broadcast operation. At this point the processors perform recursive bisection using an algorithm that is based on nested dissection and greedy partitioning refinement. However, each processor explores only a single path of the recursive bisection tree. At the end each processor stores the vertices that correspond to its partition of the p-way partitioning. Note that after the initial all-to-all broadcast operation, the algorithm proceeds without any further communication. Note that the algorithm used for computing the initial partitioning of the graph in the parallel multilevel algorithm is different from the multilevel recursive bisection used in

the serial algorithm. The multilevel algorithm produces significantly better initial partitions than nested dissection but it requires more time. Consequently, the initial partitioning step may become a bottleneck for a very large number of processors, particularly for smaller graphs. However, due to the k-way refinement performed in the uncoarsening phase, the final partitions are only slightly worse than those produced by the serial k-way algorithm (that uses the multilevel recursive bisection algorithm for computing initial partitions). Thus, the use of nested dissection for initial partitioning (in place of a more accurate multilevel recursive bisection scheme) trades a slight reduction in quality for better run time and scalability.

Parallel Uncoarsening

In the uncoarsening phase, the partitioning is projected from the coarse graph to the next-level finer graph, and it is refined using the greedy algorithm, i.e. only moving positive gain nodes . Recall that during a single phase of the refinement in the serial algorithm, vertices are randomly traversed and moved to a partition that leads to greater decrease in the edge-cut subject to the balance constraint. After each such vertex movement, the external degrees of the adjacent vertices are updated.

In the parallel formulation of greedy refinement, we retain the spirit of the serial algorithm, but we change the order in which the vertices are traversed to determine whether they can be moved to different partitions. In particular, the single phase of the refinement algorithm is broken up into c subphases, where c is the number of colors of the graph to be refined. During the i'th phase, all the vertices of color i are considered for movement, and the subset of these vertices that leads to a reduction in the edge-cut (or improves the balance without increasing the edge-cut) is moved. Since the vertices with the same color form an independent set, the total reduction in the edge-cut achieved by moving all vertices at the same time is equal to the sum of the edge-cut reductions achieved by moving these vertices one after the other. After performing this group movement, the external degrees of the vertices adjacent to this group are updated, and the next color is considered.

During the parallel refinement step, it appears natural to physically move the vertices as they change partitions. That is, each processor initially stores all the vertices of a single partition, and as vertices move between partitions during refinement, they can also move between the corresponding processors. However, in the context of multilevel graph partitioning such an approach requires significant communication. This is because, for each vertex v in the coarse graph that we move, we need to send not only the adjacency list of v but also the adjacency lists of all the vertices collapsed in v for the higher level finer graphs. In our parallel refinement algorithm we solve this problem as follows. Vertices do not move from processor to processor–only the partition number associated with each vertex changes. This also ensures that the computations performed during the

refinement are reasonably load balanced provided that the vertices are initially distributed in a random order. In this case, during refinement, each processor will have some boundary vertices that need to be moved since each processor stores a roughly equal number of vertices from all p partitions. This also leads to a simpler implementation of the parallel refinement algorithm, since vertices (and their adjacency lists) do not have to be moved around. Of course, all the vertices are moved to their proper location at the end of the partitioning algorithm, using a single all-to-all personalized communication.

The balance constraint is maintained as follows. Initially, each processor knows the weights of all p partitions. During each refinement subphase, each processor enforces balance constraints based on these partition weights. For every vertex it decides to move, it locally updates these weights. At the end of each subphase, the global partition weights are recomputed, so that each processor knows the exact weights. Even though the balance constraints maintained by this scheme are less exact than those maintained by the serial algorithm, our experiments have shown that the hybrid of local and global partition balance constraints is able to produce well-balanced partitions.

Furthermore, the above parallel refinement algorithm is highly concurrent, as long as the number of colors is small compared to the total number of vertices in the graph. For three-dimensional finite element meshes with tetrahedral elements, the number of colors tends to be less than 20, and for the graphs corresponding to their duals, it tends to be less than 5. Since the serial and parallel refinement algorithms are similar in spirit, both exhibit similar partitioning refinement capabilities.

9.3 Karlsruhe Parallel Partitioner

Although several successful multilevel partitioners have been developed in the last two decades, Holtgrewe, Sanders and Schulz had the impression that certain aspects of the method are not well understood and therefore have built our own graph partitioner KaPPa [66] (Karlsruhe Parallel Partitioner) with a focus on scalable parallelization.

Parallel Coarsening

During coarsening the algorithm uses the same distribution scheme of for the initial graph as ParMetis does, i.e. each processor receives n/p vertices. Then preliminary partition of the graph is computed, e.g., using coordinate information. Currently recursive bisection for nodes with 2D coordinates that alternately splits the data by the x-coordinate and the y-coordinate [15, 16] is implemented. We can also use the initial numbering of the nodes. Note that the initial partitioning does not

directly affect the final partitioning computed later – its main purpose is to increase locality for the computation of matchings. In addition, [66] introduced the usage of the already discussed concept of edge ratings. It uses edge ratings to rate the edges and compute a matching afterwards using those ratings.

We then combine a sequential matching algorithm running on each partition and a parallel matching algorithm running on the *gap graph*. The gap graph consists on those edges $\{u, v\}$ where u and v reside on different PEs and $\omega(\{u, v\})$ exceeds the weight of the edges that may have been matched by the local matching algorithms to u and v. The parallel matching algorithm itself iteratively matches edges that $\{u, v\}$ are locally heaviest both at u and v until no more edges can be matched (localMax algorithm for the gap graph).

Initial Partitioning

The contraction is stopped when the number of remaining nodes on some PE is below $\max(20, n/(\alpha k^2))$ for some tuning parameter α. The graph is then small enough to be partitioned on a single PE. The framework allows using pMetis or Scotch for initial partitioning. We use the sequential algorithms and run them simultaneously on all PEs, each with a different seed for the random number generator. Since initial partitioning is very fast, it is also repeated several times. The best solution is then broadcast to all PEs.

Local Search

Recall that the refinement phase iteratively uncontracts the matchings contracted during the contraction phase. After a matching is uncontracted, local search based refinement algorithms move nodes between block boundaries in order to reduce

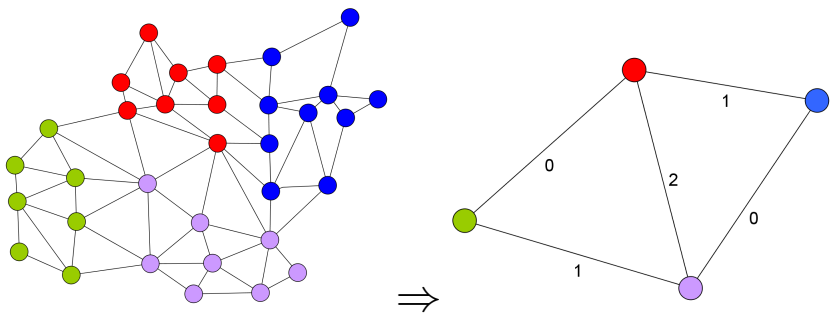


Figure 9.3: A graph which is partitioned into four blocks and its corresponding quotient graph $\mathcal{Q}$. The quotient graph has an edge coloring indicated by the numbers and each edge set induced by edges with the same color form a matching $\mathcal{M}(c)$. Pairs of blocks with the same color can be refined in parallel.

the cut while maintaining the balancing constraint. As most other current systems, we adopt the basic approach from [46] which runs in linear time. The basic idea behind our parallel refinement algorithm is that at any time, each PE may work on one pair of neighboring blocks performing a local search constrained to moving nodes between these two blocks. In order to assign pairs of blocks to PEs, we use the *quotient graph* Q whose nodes are blocks of the current partition and whose edges indicate that there are edges between these blocks in the underlying graph G. Since we have the same number of PEs and blocks, each PE will work the block assigned to it and at one of its neighbors in Q. From now on, we will therefore identify blocks and PEs. Figure 9.3 gives an example.

We use matchings of Q to define with which neighbor in Q a PE is working at a particular point in time. If u, v is in the matching, both corresponding PEs will refine the partitions u and v using different seeds for their random number generator. See Section 9.3 for more details. After the local search is finished, the better partitioning of the two blocks is adopted. Of course, for a good partition, we need to perform local search on every edge of Q eventually (we call this a *global iteration*). Section 9.3 describes our approaches for ensuring this. We ensure this by iterating through the matchings defined by an edge coloring of Q. See Section 9.3 for more details.

Overall, this approach naturally defines a nested loop controlling our local search strategy. The innermost loop moves nodes between two blocks using the FM-algorithm [46]. A *local iteration* repeats this local search. A *global iteration* iterates over the colors of an edge coloring. The loops terminate when either no improvement was found (in strong variants: when no improvement was found twice in a row.) or when a preset maximum number of iterations is exceeded.

Choosing Matchings

We have implemented two strategies. One finds edges of Q not yet used for local search in a randomized local way. The other steps through the colors of an edge coloring of the quotient graph Q. Note that this requires only local synchronization between PEs actually collaborating at a particular point in time. We only describe the latter one here since it performs slightly better in our experiments. Our coloring algorithm is a parallelization of a well known sequential greedy edge coloring algorithm: Each PE has a set $\mathcal{L}$ of free colors that have not been used for coloring incident edges. In each round of the algorithm, PEs throw a coin with sides active and passive. An active PE u picks a random incident uncolored edge $\{u, v\}$ and sends this edge together with its free-list to PE v. These *requests* are rejected if they are sent to other active PEs. Passive PEs v process requests $(\{u, v\}, \mathcal{L}')$ by choosing the color $c = \min L \cap L'$ for edge $\{u, v\}$ and sending c back to u. This algorithm is repeated until all edges are colored. It can be shown that this algorithm

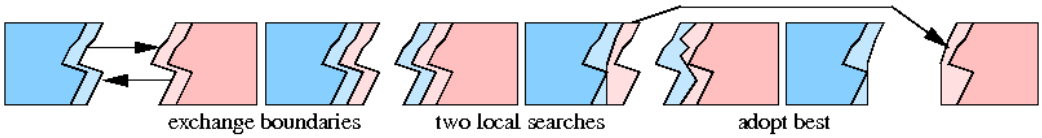


Figure 9.4: Refinement between two blocks using boundary exchange.

needs at most twice as many colors as an optimal edge coloring.

Refinement Between Two Blocks

We use a fully distributed graph data structure. More precisely, we use hybrid between a static and a dynamic graph data structure. Immediately after uncontracting a matching, every PE stores the partition it is responsible for in a static adjacency array representation (also called forward-star representation), i.e., there is an edge array storing target nodes and edge weights and a node array storing node weights and the start of the relevant segment in the edge array. In addition, we use a hash table to store migrated nodes and a second edge array for the corresponding edges. See [121] for more details. Before a local search operation, we perform a bounded breadth first search starting from the boundary of each block, and send copies of this boundary array to the partner PE in the local search. The local search is then limited to this boundary area. This way, for large graphs, only a small fraction of each block has to be communicated. If it should really happen that the local search would profit from going beyond the boundary area, this will be possible in the next iteration of some of the outer loops. Figure 9.4 shows this schematically.

The local search algorithm itself is basically the FM-algorithm [46]: For each of the two blocks A, B under consideration, a PE keeps a priority queue of nodes eligible to move. The priority is based on the *gain*, i.e., the decrease in edge cut when the node is moved to the other side. Each node is moved at most once within a single local search. The queues are initialized in random order with the nodes at the partition boundary. We have tried several queue selection strategies: *Alternating* between A and B [46], *MaxLoad* where always the heavier block gives a node, and *TopGain*, where the queue promising larger gain is used. In order to achieve a good balance, TopGain adopts the exception that MaxLoad is used when one of the blocks is overloaded. When not otherwise mentioned, we use TopGain with random tie breaking. There is also a variant *TopGainMaxLoad* that uses MaxLoad when both queues promise the same gain.

The search is broken when more than $\alpha \min\{|A|, |B|\}$ nodes have been moved without yielding an improvement. When the search stops, search is rolled back to the state with the lexicographically best value of the tuple $(\mathsf{imbalance}, \mathsf{cutValue})$. Where $\mathsf{imbalance}$ is $\max(0, \max(c(A) - L_{\max}, c(B) - L_{\max}))$.

Algorithm 13 Parallel Uncoarsening

Data: *G=(V,E)*, initial partition *P*
distribute *G* according to *P*
compute quotient graph *Q=(V_Q, E_Q)* in parallel
compute edge colouring $C : E_Q \to S$ of Q in a distributed way
for all $c \in C$ **do**
  for all $e = \{P_i, P_j\} \in M_C(c)$ **do in parallel**
    perform local two-way refinement between P_i and P_j
  end for
end for

Data Structure. During uncoarsening, a fully distributed graph data structure is used. Each PE p holds the set of vertices called block V_p. Initially, each PE keeps a static graph data structure for its own block. There is an edge array storing target nodes and edge weights and a node array storing node weights and the start of the relevant segment in the edge array. Since nodes can migrate instead of being static, in addition, a hash table is used to store migrated nodes and a second edge array for the corresponding edges. The hash table stores the begin and end pointer in E. Hashing with linear probing is most appropriate.

9.4 Parallel Cluster Contraction

Processing large complex networks like social networks or web graphs has recently attracted considerable interest. To do this in parallel, one needs to partition them into pieces of about equal size. Unfortunately, previous parallel graph partitioners originally developed for more regular mesh-like networks do not work well for these networks. This section addresses this problem by parallelizing and adapting the *label propagation* technique originally developed for graph clustering. By introducing size constraints, label propagation becomes applicable for both the coarsening and the refinement phase of multilevel graph partitioning. We obtain very high quality by applying a highly parallel evolutionary algorithm to the coarsest graph. The resulting system is both more scalable and achieves higher quality than state-of-the-art systems like ParMetis or PT-Scotch.

The main contributions of paper presented in this section [96] are a scalable parallelization of the size-constrained label propagation algorithm and an integration into a multilevel framework that enables us to partition large complex networks. The parallel size-constrained label propagation algorithm is used to compute a graph clustering which is contracted. This is repeated until the graph is small enough. The coarsest graph is then partitioned by the coarse-grained distributed

evolutionary algorithm KaFFPaE [118] (which we describe later in this script). During uncoarsening the size-constraint label propagation algorithm is used as a simple, yet effective, parallel local search algorithm. The presented scheme speeds up computations and improves solution quality on graphs that have a very irregular (but often also hierarchically clustered) structure such as social networks or web graphs.

Parallel Label Propagation

We shortly outline our parallel graph data structure and the implementation of the methods that handle communication. First of all, each PE gets a subgraph, i.e. a contiguous range of nodes $a..b$, of the whole graph as its input, such that the subgraphs combined correspond to the input graph. Each subgraph consists of the nodes with IDs from the interval $I := a..b$ and the edges incident to the nodes of those blocks, as well as the end points of edges which are not in the interval I (so-called ghost or halo nodes). This implies that each PE may have edges that connect it to another PE and the number of edges assigned to the PEs might vary significantly. The subgraphs are stored using a standard adjacency array representation, i.e. we have one array to store edges and one array for nodes storing head pointers to the edge array. However, the node array is divided into two parts. The first part stores local nodes and the second part stores ghost nodes. The method used to keep local node IDs and ghost node IDs consistent is explained in the next paragraph. Additionally, we store information about the nodes, i.e. its current block and its weight.

Instead of using the node IDs provided by the input graph (called global IDs), each PE maps those IDs to the range $0 .. n_p - 1$, where n_p is the number of distinct nodes of the subgraph. Note that this number includes the number of ghost nodes the PE has. Each global ID $i \in a .. b$ is mapped to a local node ID $i - a$. The IDs of the ghost nodes are mapped to the remaining $n_p - (b - a)$ local IDs in the order in which they appeared during the construction of the graph structure. Transforming a local node ID to a global ID or vice versa, can be done by adding or subtracting a. We store the global ID of the ghost nodes in an extra array and use a hash table to transform global IDs of ghost nodes to their corresponding local IDs. Additionally, we store for each ghost node the ID of the corresponding PE, using an array for $\mathcal{O}(1)$ lookups.

To parallelize the label propagation algorithm, each PE performs the algorithm on its part of the graph. Recall, when we visit a node v, it is moved to the block that has the strongest eligible connection. Note that the cluster IDs of a node can be arbitrarily distributed in the range $0 .. n-1$ so that we use a hash map to identify the cluster with the strongest connection. Since we know that the number of distinct neighboring cluster IDs is bounded by the maximum degree in the graph, we use

hashing with linear probing. At this particular point of the algorithm, hashing with linear probing is much faster than using the hash map of the STL.

During the course of the algorithm, local nodes can change their block and hence the blocks in which ghost nodes are contained can change as well. Since communication is expensive, we do not want to perform communication each time a node changes its block. We use the following scheme to *overlap* communication and computation. The scheme is organized in phases. We call a node *interface node* if it is adjacent to at least one ghost node. The PE associated with the ghost node is called adjacent PE. Each PE stores a separate send buffer for all adjacent PEs. During each phase, we store the block ID of interface nodes that have changed into the send buffer of each adjacent PE of this node. Communication is then implemented asynchronously. In phase κ, we send the current updates to the adjacent PEs and receive the updates of the adjacent PEs from round $\kappa - 1$, for $\kappa > 1$. Note that in case the label propagation algorithm has converged, i.e. no node changes its block any more, the communication volume is really small.

The degree-based node ordering approach of the label propagation algorithm that is used during coarsening is parallelized by considering only the local nodes for this ordering. In other words, the ordering in which the nodes are traversed on a PE is determined by the node degrees of the local nodes of this PE. During uncoarsening random node ordering is used.

Note that due to the parallelization it is possible that oscillations occur. Overlapping computation and communication reduces this effect. On the other hand, we do not need an optimal clustering so that we tolerate oscillations, stop prematurely and still get good quality.

Balance/Size Constraint

Maintaining the balance of blocks is more difficult in the parallel case than in the sequential case. We use two different approaches to maintain balance, one for coarsening and the other one for uncoarsening. The reason for this is that during coarsening there is a large number of blocks and the constraint is rather soft ($\frac{L_{\max}}{f}$), whereas during uncoarsening the number of blocks is small and the constraint is tight ($L_{\max}$).

We maintain the balance of different blocks *during coarsening* as follows. Roughly speaking, a PE maintains and updates only the local amount of node weight of the blocks of its local and ghost nodes. Due to the way the label propagation algorithm is initialized, each PE knows the exact weights of the blocks of local nodes and ghost nodes in the beginning. Label propagation then uses the local information to bound the block weights. Once a node changes its block, the local block weight is updated. Note that this does not involve additional communication. We decided to use this localized approach since the balance

constraint is not tight during coarsening. More precisely, the bound on the cluster sizes during coarsening is a tuning parameter and the overall performance of the system does not directly depend on the exact choice of the parameter.

During uncoarsening we use a different approach since the number of blocks is much smaller and it is unlikely that the previous approach yields a feasible partition in the end. This approach is similar to the approach that is used within ParMetis [76]. Initially, the exact block weights of all k blocks are computed locally. The local block weights are then aggregated and broadcast to all PEs. Both can be done using one allreduce operation. Now each PE knows the global block weights of all k blocks. The label propagation algorithm then uses this information and locally updates the weights. For each block, a PE maintains and updates the total amount of node weight that local nodes contribute to the block weights. Using this information, one can restore the exact block weights with one allreduce operation which is done at the end of each computation phase. This approach would not be feasible during coarsening as there are n blocks in the beginning of the algorithm and each PE holds the block weights of all blocks.

Parallel Contraction and Uncoarsening

The *parallel contraction* algorithm works as follows. After the parallel size-constrained label propagation algorithm has been performed, each node is assigned to a cluster. Recall the definition of our general contraction scheme. Each of the clusters of the graph corresponds to a coarse node in the coarse graph and the weight of this node is set to the total weight of the nodes that are in that cluster. Moreover, there is an edge between two coarse nodes iff there is an edge between the respective clusters and the weight of this edge is set to the total weight of the edges that run between these clusters in the original graph.

In the parallel scheme, the IDs of the clusters on a PE can be arbitrarily distributed in the interval $0 \,..\, n-1$, where n is the total number of nodes of the input graph of the current level. Consequently, we start the parallel contraction algorithm by finding the number of distinct cluster IDs which is also the number of coarse nodes. To do so, a PE p is assigned to count the number of distinct cluster IDs in the interval $I_p := p\lceil \frac{n}{P} \rceil + 1 \,..\, (p+1)\lceil \frac{n}{P} \rceil$, where P is the total number of PEs used. That means each PE p iterates over its local nodes, collects cluster IDs a that are not local, i.e. $a \notin I_p$, and then sends the non-local cluster IDs to the responsible PEs. Afterwards, a PE counts the number of distinct local cluster IDs so that the number of global distinct cluster IDs can be derived easily by using a reduce operation.

Let n' be the global number of distinct cluster IDs. Recall that this is also the number of coarse nodes after the contraction has been performed. The next step in the parallel contraction algorithm is to compute a mapping $q : 0 \,..\, n-1 \rightarrow 0 \,..\, n'-1$

which maps the current cluster IDs to a contiguous interval over all PEs. This mapping can be easily computed in parallel by computing a prefix sum over the number of distinct local cluster IDs a PE has. Once this is done, we compute the mapping $C : 0 .. n - 1 \rightarrow 0 .. n' - 1$ which maps a node ID of G to its coarse representative. Note that, if a node v is in cluster V_ℓ after the label propagation algorithm has converged, then $C(v) = q(\ell)$. After computing this information locally, we also propagate the necessary parts of the mapping to neighboring PEs so that we also know the coarse representative of each ghost node. When the contraction algorithm is fully completed, PE p will be *responsible* for the subgraph $p\lceil \frac{n'}{P} \rceil + 1 .. (p+1)\lceil \frac{n'}{P} \rceil$ of the coarse graph. To construct the final coarse graph, we first construct the weighted quotient graph of the local subgraph of G using hashing. Afterwards, each PE sends an edge (u, v) of the local quotient graph, including its weight and the weight of its source node, to the responsible PE. After all edges are received, a PE can construct its coarse subgraph locally.

The implementation of the *parallel uncoarsening* algorithm is simple. Each PE knows the coarse node for all its nodes in its subgraph (through the mapping C). Hence, a PE requests the block ID of a coarse representative of a fine node from the PE that holds the respective coarse node.

Iterated Multilevel Schemes

Iterated V-cycles are also used within clustering-based coarsening in our previous work [95]. To adapt the iterated multilevel technique for this coarsening scheme, it has to be ensured that cut edges are not contracted after the first multilevel iteration. This is done by modifying the label propagation algorithm such that each cluster of the computed clustering is a subset of a block of the input partition. In other words, each cluster only contains nodes of one unique block of the input partition. Hence, when contracting the clustering, every cut edge of the input partition will remain. Recall that the label propagation algorithm initially puts each node in its own block so that in the beginning of the algorithm each cluster is a subset of one unique block of the input partition. This property is kept during the course of the label propagation algorithm by restricting the movements of the label propagation algorithm, i.e. we move a node to an eligible cluster with the strongest connection in its neighborhood that is in the same block of the input partition as the node itself. We do the same in our parallel approach to realize V-cycles.

The Overall Parallel System

The overall parallel system works as follows. We use ℓ iterations of the parallel size-constrained label propagation algorithm to compute graph clusterings and contract them in parallel. We do this recursively until the remaining graph has less

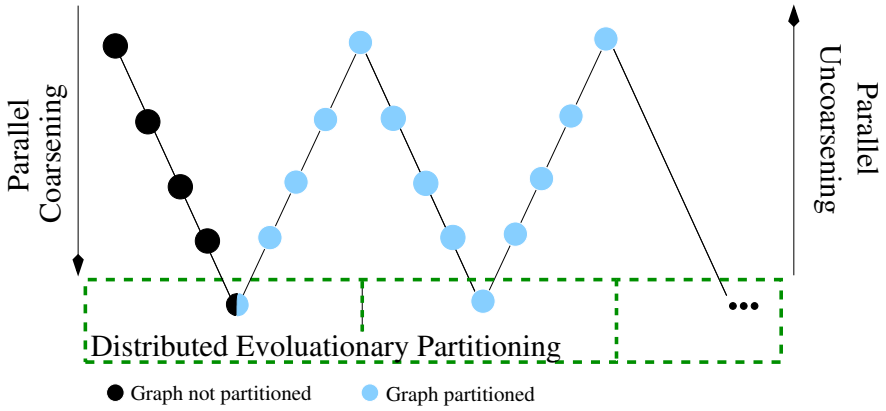


Figure 9.5: The overall parallel system. It uses the parallel cluster coarsening algorithm, the coarse-grained distributed evolutionary algorithm KaFFPaE to partition the coarsest graph and parallel uncoarsening/local search. After the first iteration of the multilevel scheme the input partition is used as a partition of the coarsest graph and used as a starting point by the evolutionary algorithm.

than $20\,000$ nodes left. The distributed coarse graph is then collected on each PE, i.e. each PE has a copy of the complete coarsest graph. We use this graph as input to the coarse-grained distributed evolutionary algorithm KaFFPaE, to obtain a high quality k-partition of it. We modified KaFFPaE to use combine operations that also use the clustering-based coarsening scheme from above. The best solution of the evolutionary algorithm is then broadcast to all PEs which transfer the solution to their local part of the distributed coarse graph. Afterwards, we use the parallel uncoarsening algorithm to transfer the solution of the current level to the next finer level and apply r iterations of the parallel label propagation algorithm with the size constraints of the original partitioning problem (setting $W = (1+\epsilon)\lceil\frac{|V|}{k}\rceil$) to improve the solution on the current level. We do this on each level of the hierarchy and obtain a good partition of the input network in the end. If we use iterated V-cycles, we use the given partition of the coarse graph as input to the evolutionary algorithm. More precisely, one individual of the population is the input partition on each PE. This way it is ensured that the evolutionary algorithm computes a partition that is at least as good as the given partition. Note that our initial partitioner is usually able to compute partitions that fulfill the desired balance constraint on the coarsest level. Hence, to ensure that the final partition of our parallel algorithm is balanced, we do not perform any parallel local search during the last V-cycle. A schematic of the overall system is shown in Figure 9.5.

9.5 Facebook's Approach

The system that has been used by Facebook [133] works as follows. First of all, the system is implemented in Giraph, a "Think like a Vertex" framework. That means the algorithm is described from a vertex point of view.

To find initial blocks of the partitioning, their algorithm either uses random initialization, e.g. each vertex picks a random block, or better, apply a greedy geographic assignment. With the latter, it is possible to effectively achieve convergence within a single step of the update algorithm, while random requires many iterations until convergence and yields a solution of lesser quality.

Afterwards, label propagation is used to relocate inefficiently assigned nodes while respecting strict block balancing constraints, using so called *Balanced Label Propagation*. More precisely, every node determines where it would prefer to move, and how much each node would gain from its preferred relocation. Secondly, node gains are sorted for each block pair and a *Constrained Relocation Linear Program* is constructed. Subsequently, the program is solved, which determines how many and which nodes should be moved between each pair of blocks. The movements are then executed. Note that the preferences of the nodes are collected and that the linear program is solved on a single machine. Overall, this constitutes one iteration. Compared to ordinary label propagation, the difference is that rather than moving every node that asks to move, the algorithm pauses, solves a linear program, and then proceeds to move as many nodes as possible without breaking the balance.

Constraint Relocation Linear Problem. Given an initial feasible partitioning, we wish to maintain the specified balance of nodes across blocks between iterations. The key challenge is however that some shards will be more popular than others. In fact, under ordinary label propagation without any balance constraints, labelling all nodes with the same single label is a trivial equilibrium. Because we won't be able to move all nodes, the greedy approach is to synchronously move those nodes that stand to increase their colocation count (the number of graph neighbors they are co-located with) the most.

Consider the nodes that are assigned to block i but would prefer to be in block j (positive gain). Order these nodes according to their gain, from greatest gain to smallest gain, labelling them $k = 1, .., K$. Let $u_{ij}(k)$ be the change in utility from moving the k-th node from block i to j. Let $f_{ij}(x) = \sum_{k=1}^{x} u_{ij}(k)$ be the relocation utility function between shard i and j, the total utility gained from moving the leading x nodes from i to j. Observe that because $u_{ij}(k) \geq 0$ and $u_{ij}(k) \geq u_{ij}(k+1)$ for all k, all $f_{ij}(x)$ are increasing and concave. The goal can be formulated as a concave utility maximization problem with linear constraints.

Problem formulation Given a graph $G = (V, E)$ with the node set partitioned into k blocks $V_1, ..., V_k$, and size constraints $S_i \leq |V_i| \leq T_i, \forall i$, the constrained relocation problem is to maximize:

$$\max \sum_{i,j} f_{ij}(x_{ij}) \text{ s.t.}$$

$$S_i - |V_i| \leq \sum_{j \neq i}(x_{ij} - x_{ji}) \leq T_i - |V_i|, \forall i$$

$$0 \leq x_{ij} \leq P_{ij}, \forall i, j$$

with P_{ij} is the number of nodes that desire to move from V_i to V_j, and $f_{ij}(x)$ is the relocation utility function between V_i and V_j. For continuous values of x_{ij}, the above problem can be reformulated into a tractable optimization problem. Such a relaxation on all x_{ij} is fully reasonable as long as the number of nodes seeking to be moved between each par of shards is large. In the above problem formulation, f_{ij} are piece-wise linear concave functions. This can be easily seen when considering the slopes of f_{ij}, which are constant across all intervals where the ordered users have the same derived utility.

Theorem 21
Assume a bounded degree graph G. Then $f(x) = \sum_{i,j} f_{ij}(x_{ij})$, is a piece-wise-linear concave function and separable in x_{ij}.

Proof. The separability is clear from the fact that the f_{ij} depend on different variables. Since users are atomic and the graph is of bounded degree, there are a finite number of utilities, and the sum is therefore also piece-wise linear. Since the nodes are sorted in order of decreasing utility, the function is concave. □

Theorem 22
Any piece-wise linear concave function $f(x) : R^n \to R$ can be written as $f(x) = min_{k=1,...,l}(a_k^T x + b_k)$, for some choices of a_k's and b_k's. In our case, all the a_k's and b_k's are scalar.

Theorem 23
Let $x \in R^n$ and $f(x) = min_{k=1,...,l}(a_k^T x + b_k)$ be a piece-wise linear concave function. Maximizing f(x) subject to $Ax \leq b$ is then equivalent to:

$$\max z \text{ s.t. } Ax \leq b \text{ and } a_k^T + b_k \geq z, \forall k$$

Utilizing the last two lemmas, one can thus solve the concave maximization problem in problem using a linear program.

Theorem 24
Consider a bounded degree graph $G = (V, E)$. Under continuous x_{ij}, the constrained relocation problem can be written as

$$\max \sum_{i,j} z_{ij} \text{ s.t.}$$

$$S_i - |V_i| \leq \sum_{j \neq i} (x_{ij} - x_{ji}) \leq T_i - |V_i|, \forall i$$

$$0 \leq x_{ij} \leq P_{ij}, \forall i, j$$

$$-a_{ijk} x_{ij} + z_{ij} \leq b_{ijk}, \forall i, j, k$$

where all a_{ijk} and b_{ijk} derive directly from the relocation utility functions f_{ij}. Assuming n shards and at most K unique utility gains achieved by nodes that would like to move, this constitutes a linear program with $2k(k-1)$ variables and at most $2k^2 + Kk(k-1)$ sparse constraints.

In order to reduce the number of constraints, the following observation can be useful. For each pair of blocks, the handful of nodes that stand to gain the most are likely to be contribute relatively unique utility levels, and so contribute many of the constraints in the problem. This is rather unnecessary, since those nodes are highly likely to move. Thus, by disregarding the unique utility levels of the first C nodes, all very likely to move, and approximating them by the mean gain of this population, we can greatly reduce the number of constraints.

References

Meyerhenke, Sanders and Schulz [95, 96], Simon [127], Berger, Bokhari [17], Karypis and Kumar [76], Holtgrewe, Sanders Schulz [66]and Schulz[122]. This chapter was created by Christoph Hess, Matthias Stumpp and Christian Schulz.

Chapter 10

Semi-External and External GP

To be able to process huge unstructured networks on cheap commodity machines one can rely on graph partitioning and partition the graph under consideration into a number of blocks such that each block fits into the internal memory of the machine while edges running between blocks are minimized (see for example [86]). However, to do so the partitioning algorithm itself has to be able to partition networks that do not fit into the internal memory of a machine. In this chapter, we present semi-external and external algorithms for the graph partitioning problem that are able to compute high quality solutions. The chapter is based on [3].

10.1 Computational Models

We look at two models: the external and the semi-external model [2]. In both models, one wants to minimize the number of I/O operations. In the external model it is assumed that the graph does not fit into internal memory whereas the semi-external memory assumes that there is enough memory for the nodes of the graph to fit into internal memory, but not enough for the edges. We will use the following notations: M is the size of internal memory, B is the size of a disk block, $\mathcal{O}(\frac{N}{B}) = \text{Scan}(N)$ is the number of I/O operations needed for reading or writing an array of size N and $\mathcal{O}(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}) = \text{Sort}(N)$ is the number of I/O operations needed for sorting an array of size N.

Graph Data Structure

To store a graph in external memory, we use a data structure similar to an adjacency array. This data structure allows us to inspect all edges using $\text{Scan}(|E|)$ I/O operations. An external array of the edges contains the adjacency lists of each node in increasing order of their IDs. Each element of the adjacency list of a node u is a

pair (v, w), where v is the target of the edge (u, v) and $w = w(u, v)$ is the weight of the edge. We mark the end of each adjacency list by using a sentinel pair. This allows us to determine easily if we reached the end of the adjacency list of the node that we currently process. The second external array stores node offsets, i.e. for each node we store a pointer to the beginning of its adjacency list in the edge array. The third external array contains the weights of the nodes.

10.2 (Semi-)External Graph Clustering

We now explain how graph clusterings can be obtained in both, the semi-external and the external memory model. We present multiple algorithms: a semi-external LPA that can deal with size-constraints, an external LPA that does not use size-constraints, as well as a coloring-based graph clustering algorithm inspired by label propagation that can also maintain size-constraints in the external model.

Label Propagation

Recall how label propagation works: In the beginning each node belongs to its own cluster. Afterwards the algorithm works in rounds. In each round, the algorithm visits each node in increasing order of their IDs. When a node v is visited, it is *moved* to the block that has the strongest connection to v, i.e. it is moved to the cluster V_i that maximizes $\omega(\{(v, u) \mid u \in N(v) \cap V_i\})$. Ties are broken randomly. If the algorithm is used to compute a size-constrained clustering, the selection rule is modified such that only moves are eligible that do not result in overloaded blocks. Suppose the algorithm is currently processing a node v. Now we scan the adjacency list of the node v in the respective memory model and compute the new cluster of this node. If a size-constraint is present, we also need a scheme to manage the sizes of each cluster/block.

Semi-External Label Propagation. This is the simple case: since we have $\mathcal{O}(|V|)$ internal memory, we can afford to store the cluster IDs in internal memory. Additionally, we maintain an array of size $|V|$ in internal memory that stores the cluster sizes. Hence, one iteration of the semi-external LP algorithm can be done using Scan$(|E|)$ I/O operations.

Parallel Semi-External Label Propagation. Recall that the LP algorithm iterates through the external array of edges. In attempt to accelerate semi-external LPA and to get closer to the I/O bound, we parallelize the processing of a disk block of edges. Since, the LP algorithm processes nodes interdependently, we divide the disk block into equal ranges and process them in parallel. We now

explain how we process nodes which have adjacency lists that belong to different disk blocks (see Figure 10.1 for an example). Each thread t begins to process its range $[begin_t; end_t)$ of the disk block. Afterwards, the range is shifted such that each adjacency list in the block is processed by precisely one thread. Consider the example depicted in Figure 10.1. Here, the thread finds the end of the adjacency list 1 in $[begin_t; end_t)$ and iterates through the elements until the end of adjacency list 2 is reached. The colored area in Figure 10.1 represents the range that will be actually processed by thread t.

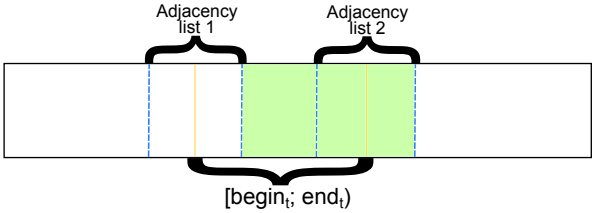


Figure 10.1: Range processed by a thread

To maintain up-to-date cluster sizes, we do not move a node immediately, i.e. we store the moves which were generated by the threads during the processing of the disk block. Afterwards, all moves are processed sequentially and we make a move if it does not violate the size constraint.

External Label Propagation. To propagate the cluster IDs of adjacent nodes, we use time forward processing [137]. More precisely, we maintain two external priority queues [116]: one for the current and one for the next round. Initially, the current queue contains triples (v, c, w) for each edge $(u, v) \in E : v < u$ where v is the key value, w denotes the weight of the edge and c is the current cluster ID of u. When the algorithm scans a node u, all triples (u, c, w) are on top of the priority queue since the nodes are processed in increasing order of their ID. The tuples are then extracted using the operations Pop and Top. This means, we know the current cluster ID of all adjacent nodes of u and can calculate the new cluster ID. When the new cluster ID is computed, the algorithm pushes triples with the new cluster ID for all adjacent nodes into the next and current priority queue depending on the node ID of the neighbor v: if $u < v$ we push $(v, \text{cluster}[u], w(u, v))$ to the current priority queue and if $v < u$ we push it to the next priority queue. At the end of a round we swap the priority queues.

Each operation of the priority queue Pop, Push and Top is called $\mathcal{O}(|E|)$ times. Each of the operations can be done using $\mathcal{O}(\frac{1}{B} \log_{\frac{M}{B}} \frac{|E|}{B})$ I/O operations amortized [116, 6]. Thus, the overall algorithm uses $\mathcal{O}(\frac{|E|}{B} \log_{\frac{M}{B}} \frac{|E|}{B}) = \text{Sort}(|E|)$ I/O operations.

Coloring-based Graph Clustering

We now present another approach to compute a graph clustering in the external model. The algorithm is able to maintain the sizes of all clusters in the external memory model. The *main idea* of the algorithm is to process *independent sets*. Due to the definition of an independent set, a change of the cluster ID of a node will not have an effect on the other nodes within the set. However, the changes of all adjacent nodes of v need to be taken into account.

Assume that we have a node coloring $C = \{C_1, C_2, \ldots, C_\ell\}$ of the graph, where C_i is the set of vertices with the same color i. Note that each set C_i forms an independent set. For each set C_i, we maintain an external array (bucket) of tuples $\mathcal{T}_i$ and allocate a buffer of size B internal memory for each bucket $\mathcal{T}_i$. Hence, we assume that the number of colors $|C|$ is smaller than $\frac{M}{B}$.

The bucket clustering algorithm also works in rounds. Roughly speaking, in each round it processes the buckets in increasing order of their color and updates the cluster IDs of all nodes. When we process a bucket we need the cluster IDs of all adjacent nodes. Hence, we define the content of a bucket as follows: we store tuples so that we know the current cluster ID of each adjacent neighbor, its color, ID of the neighbor and the weight of the edge. More precisely, initially for a node u, we store the following tuples for all adjacent nodes v with color$[v] <$ color$[u]$ in the corresponding buckets $\mathcal{T}_{\text{color}[v]}$: $(v, \text{cluster}[u], u, w(u,v), \text{color}[u])$. To do this efficiently, we augment the array of edges by adding the color of the target node v to each edge (u, v) before the initialization step. Note that these tuples contain the complete information about the graph structure and that the information suffices to update the cluster of a node.

When the algorithm processes a bucket $\mathcal{T}_i$, it sorts the elements of the bucket lexicographically by the first and second component. Afterwards, it scans the tuples of the bucket and calculates a new cluster ID for each node in C_i in the same manner as the LPA. When the bucket is processed, we push tuples with the new cluster IDs to the corresponding bucket, i.e. for each tuple $(v, \text{cluster}[u], u, w(v,u), \text{color}[u])$ in bucket $\mathcal{T}_i$, we push the tuple $(u, \text{cluster}[v], v, w(u,v), \text{color}[v])$ into the bucket $\mathcal{T}_{\text{color}[u]}$.

Lemma 25
Processing a bucket $\mathcal{T}$ can be done in Sort($|\mathcal{T}|$) I/O-operations.

Proof. For sorting the bucket, we need Sort($|\mathcal{T}|$) I/O operations. We need Scan($|\mathcal{T}|$) I/O operations for scanning the bucket. Hence, the algorithm uses Sort($|\mathcal{T}|$) I/O operations.

Theorem 26
The bucket algorithm requires Sort($|E|$) I/O-operations for one iteration of label propagation.

Proof. Adding the information about the colors of target nodes to the edges requires Sort($|E|$) I/O operations. Our bucket initialization uses Scan($|E|$) I/O operations. Suppose we have the buckets $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$. All buckets can be processed using Sort($|\mathcal{T}_1|$) + ... + Sort($|\mathcal{T}_\ell|$) = Sort($|E|$) I/O operations. Overall, we use Scan($|E|$) I/O operations for pushing tuples with the new cluster IDs into the respective buckets. Hence, the algorithm can be implemented using Sort($|E|$) I/O operations.

Graph Coloring. Computing the graph coloring is a very important part of the bucket graph clustering algorithm. Note that the number of colors is equal to the number of buckets and we want to maintain as few buckets as possible. This is due to the fact that we need an amount of B space for each bucket in internal memory. Moreover, the size of each bucket must be smaller than an upper bound, since each bucket has to fit into internal memory during our experiments. To compute a coloring, we use the time forward processing technique [137] with an additional size-constraint on the color classes that can be maintained in internal memory. This allows us to build a coloring using Sort($|E|$) I/O operations. Note that the coloring is computed only once so that the cost for computing the coloring can be amortized over many iterations of label propagation.

External Graph Clustering Algorithm with Size-Constraints. In this section we describe how we modify the coloring-based clustering algorithm, so that it can handle a size-constraint. The main advantage of the coloring-based clustering algorithm is as follows. When we process a bucket, the cluster IDs of all adjacent nodes will not change. This allows us to maintain a data structure with up-to-date sizes of the clusters of the nodes of the independent set and their neighbors. In the following, we consider two different data structures depending on if each of the buckets fits into internal memory or not. In both cases, we use an external array that stores the sizes of all clusters. We start by explaining the case where each bucket fits into internal memory.

Case A: each bucket fits into internal memory. In this case, we can use a hash table $\mathcal{H}$ to maintain the cluster sizes of the current bucket. The key of $\mathcal{H}$ is the cluster ID and the value is the current size of the cluster. When we process a bucket $\mathcal{T}_i$, the hash table $\mathcal{H}$ can be built as follows. We collect all cluster IDs of the nodes of the current independent set as well as their neighbors, sort them and then

iterate through the external array to get the current cluster sizes. After finishing to calculate a new cluster ID for each node in C_i, we write the updated cluster sizes to the external array. Hence, the cluster sizes are up-to-date after we processed the current bucket.

Theorem 27
The coloring-based clustering algorithm with size-constraints uses $t \cdot \mathit{Scan}(|V|) + \mathit{Sort}(|E|)$ *I/O operations, where* $t = \max(\frac{|E|}{M}, |C|)$ *is the amount of buckets such that each bucket fits into internal memory.*

Proof. First, we prove the complexity to create the data structure containing the clusters sizes. In the worst case, each tuple $(v, \text{cluster}[u], u, w(v,u), \text{color}[u])$ of the bucket $\mathcal{T}$ has a unique cluster ID and also the cluster IDs of each node v is unique. Hence, we need an additional amount of $\mathcal{O}(|\mathcal{T}|)$ internal memory for the hash table. For the saving and sorting the cluster IDs from the bucket we use $\mathcal{O}(1)$ I/O operations since we have the bucket in internal memory. Reading and writing the sizes of the clusters to and from the external array uses Scan($|V|$) I/O-operations.

Now we estimate the amount of buckets that fit into internal memory. There are two cases. If a bucket does not fit into internal memory, we need to divide it into multiple buckets. Since the overall size of all buckets is $\mathcal{O}(|E|)$ the minimum amount of buckets (such that each fits into internal memory) is $\mathcal{O}(\frac{|E|}{M})$. Otherwise, if all buckets fit into internal memory, we have $|C|$ buckets. Since we want each bucket to fit into internal memory, we have $\max(\frac{|E|}{M}, |C|)$ buckets.

The overall I/O-volume is estimated as follows: for each bucket we need Scan($|V|$) additional I/O-operations to create the data structure containing the sizes of the clusters. There are at most $\max(\frac{|E|}{M}, |C|)$ buckets. Hence, the total number of I/O-operations is $\max(\frac{|E|}{M}, |C|) \cdot \text{Scan}(|V|) + \text{Sort}(|E|)$ (to perform the main part of the bucket clustering algorithm).

Case B: there is at least one bucket that does not fit into internal memory. This case is somewhat more complicated, since we cannot afford to store the hash table in internal memory. Basically, when we process a bucket $\mathcal{T}_i$, we do not use a hash table but an external priority queue and additional data structures which contain enough information to manage the cluster sizes. More precisely, we define a structure $\mathcal{M}$ that tells us for each node which nodes need the updated cluster size information. Nodes from C_i are still processed in increasing order of their IDs. We now explain the structures in detail.

For a node v in the current independent set C_i, let $C(v) := \{\text{cluster}[u] \mid (v,u) \in E\} \cup \{\text{cluster}[v]\}$ denote the set of adjacent clusters. An example is shown in Figure 10.2. These are the clusters that can possibly change their size

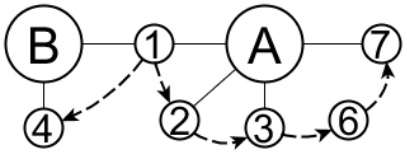


Figure 10.2: In this example, we have $C(1) = \{A, B, \text{cluster}[1]\}$ and $C(4) = \{B, \text{cluster}[4]\}$. Dotted lines denote forwarding cluster size changes. A, B are cluster IDs, 1-7 denote node ID. Node 6 belongs to cluster A. The sets of adjacent nodes for the cluster A and B are $N_A = \{1, 2, 3, 6, 7\}$ and $N_B = \{1, 4\}$. Moreover, $M_1 = \{(2, A), (4, B)\}$, $M_2 = \{(3, A)\}$.

if v changes its cluster. We now need to find all nodes from the independent set that are adjacent to the clusters or are contained in this cluster because they need to receive the updated cluster size.

The first additional data structure contains only nodes of the independent set. It is needed to build the next data structure $\mathcal{M}$. Let $N_c := \{u \in C_i \mid \exists (u, v) \in E : \text{cluster}[v] = c\} \cup \{u \in C_i \mid \text{cluster}[u] = c\}$ be the set of adjacent nodes for a cluster c that are in the current independent set (including the nodes that are in the cluster). We sort N_c in increasing order of node IDs and remove repeated elements. For a cluster c, the set N_c contains all nodes from the independent set that are adjacent to the cluster. Moreover, the order in N_c is similar to the processing order of the independent set. We denote the j-th node of N_c as N_c^j. The second additional data structure uses the first one and is defined as the set $M_v := \{(u, c) \mid c \in C(v),\ N_c^j = v,\ N_c^{j+1} = u\}$. It contains the nodes to which the node v must forward information about the changes in the cluster sizes. Roughly speaking, for each cluster in the neighborhood of v (including the cluster of v), M_v contains the adjacent node of the cluster that will be processed next. This way the information can be propagated easily. An example is shown in Figure 10.2. We now explain the details of the algorithm when processing one bucket. First, we compute the sets N_c. To do so, we build a list N of pairs (c, v) that are sorted lexicographically by their first and second component, where c is the ID of the cluster adjacent to v. For building this list, we iterate through the bucket and add pairs (c, v) for each tuple $(v, \ldots) \in \mathcal{T}_i$ to the list and also add the pair $(\text{cluster}[v], v)$ $\forall v \in C_i$. Then we sort these pairs and we are done. Note that $|N| = |C_i| + |\mathcal{T}_i| = \mathcal{O}(|\mathcal{T}_i|)$. To compute the sets M_v, we build a list $\mathcal{M}$ of triples (v, c, u), where v is the node ID and $(u, c) \in M_v$. For each $N_c^j = v$ and $N_c^{j+1} = u$ the triple (v, c, u) is added to the list. Afterwards, the triples are sorted by the first component. Note that the size of the list is at most $\mathcal{O}(|\mathcal{T}_i|)$.

Recall, that the set M_v contains the nodes that have to receive the changes in the cluster sizes. To forward the information, we use an external priority queue.

The priority queue contains triples (v, c, sz), where v is the node ID which also serves as key value, $c \in C(v)$ is the cluster and sz the size of the cluster. We initialize the priority queue as follows: we iterate through the sets N_c and put the tuples (v_1, c, sz) in the priority queue, where v_1 is the first node in N_c. The sizes of the clusters are obtained from the external array containing the cluster sizes. Then the nodes are processed. After node v is processed, we put (u, c, sz) in the priority queue for each pair $(u, c) \in M_v$. After we processed a bucket, we update the cluster sizes in the external array.

Lemma 28
When node v is processed there is a triple (v, c, sz) for each adjacent cluster on the top of priority queue with up-to-date cluster sizes.

Proof. Consider a cluster ID c and let the list N_c be $\{v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_k\}$. In the beginning, there is triple $(v_1, c, size)$ in the priority queue with the actual size of the cluster c due way the priority queue is initialized. When we process v_i, we put the triple with the updated size into the priority queue for *the* pair (v_{i+1}, c) in the list M_{v_i}. Hence, when we process v_{i+1} the size of the cluster will also be up-to-date (since it is the next adjacent node of cluster c being processed). When v is processed the triples with key value v are on top of the priority queue, due to the increasing processing order.

Lemma 29
Processing a bucket and maintaining cluster sizes costs Sort($|\mathcal{T}|$) I/O-operations.

Proof. We need Sort($|\mathcal{T}|$) I/O-operations to sort the bucket and to build the lists N and $\mathcal{M}$. The operations pop and push of the priority queue have amortized $\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B}$ cost. The number of push (or pop) operations is equal to $|\mathcal{M}|$. This is due to the fact that each element of this list means that two nodes in the bucket have to take the size of the same cluster into account. Thus, we need to forward the information from the first to the second node. This means that the total cost of all operations is Sort($|\mathcal{T}|$). Iterating through the bucket costs Scan($|\mathcal{T}|$). Hence, processing a bucket costs Sort($|\mathcal{T}|$) in total.

Theorem 30
One iteration of the coloring-based clustering algorithm costs Sort($|E|$) + $|C| \cdot$ Scan($|V|$) operations, where $|C|$ is the number of buckets.

Proof. We have $|C|$ buckets. Processing a bucket costs Sort($|\mathcal{T}|$) I/Os. The overall number of elements in the buckets is $\mathcal{O}(|E|)$. Hence, processing all buckets takes Sort($|E|$) I/O-operations. After we processed a bucket, we update the sizes of the clusters that changed during processing. This takes Scan($|V|$) I/O-operations. Thus, one iteration of the bucket clustering algorithm costs Sort($|E|$) + $|C| \cdot$ Scan($|V|$) I/O-operations.

10.3 (Semi-)External MGP

We now explain the (semi-)external multilevel graph partitioning algorithm.

Coarsening/Contraction

We have two different algorithms to create graph hierarchies depending on the memory model that we use. In general to create a graph hierarchy, we compute a clustering with size-constraints of the current graph using some algorithm from Section 10.2. The next step is to renumber the cluster IDs. The external algorithm sorts the nodes by their cluster ID and scans the sorted array assigning new cluster IDs from $0, \ldots, n' - 1$, where n' is the number of the distinct clusters. This step can be done using Sort($|V|$) I/O operations. In contrast, the semi-external algorithm uses an additional array of size $\mathcal{O}(|V|)$ to assign new cluster IDs. Hence, it needs $\mathcal{O}(1)$ I/O operations.

The external algorithm builds an array of triples $(\text{cluster}[u], \text{cluster}[v], w(u, v))$ for each edge $(u, v) \in E$ to build the contracted graph. This array is sorted lexicographically by the first two entries using Sort($|E|$) I/Os. Then we merge parallel edges and build the edges of the quotient graph by iterating through the sorted array using Scan($|E|$) I/Os. The total I/O volume of this step is Sort($|E|$). The semi-external algorithm stores pairs $(\text{cluster}[u], \text{cluster}[v])$ for each edge (u, v) in a hash table and uses it to build the contracted graph. This can be done using Scan($|E|$) I/Os. If the number of edges of the contracted graphs decreases geometrically and a constant number of label propagation iterations is assumed, the complete hierarchy can be built using Scan($|E|$) I/Os using the semi-external algorithm or Sort($|E|$) I/Os using the external algorithm.

Uncoarsening/Solution Transfer

In this step, we want to transfer a solution of a coarse level to the next finer level in the hierarchy and perform some local search. Let $\mathcal{Q} = (V_\mathcal{Q}, E_\mathcal{Q})$ be a contracted graph of the next finer level $G = (V, E)$ and let $\text{cluster}_\mathcal{Q}$ be a partition of the contracted graph. Recall that the contracted graph has been built according to a clustering of the graph G. Also note that cluster i of G corresponds to node i in the contracted graph. Hence, for a node $v \in V$ the transferred cluster ID of the coarse level is $\text{cluster}'_G[v] := \text{cluster}_\mathcal{Q}[\text{cluster}_G[v]]$.

To transfer the solution in external algorithm, we build an array of pairs $(\text{cluster}_G[v], v)$ and sort it by the first component using Sort($|V|$) I/O operations. Now we iterate through both of the arrays $\text{cluster}_\mathcal{Q}$ and $\{(\text{cluster}_G[v], v)\}$ at the same time and generate the array $\{(\text{cluster}_\mathcal{Q}[\text{cluster}_G[v]], v)\}$ which contains the transferred solution. We sort the resulting array by the second component and

apply the clustering to our graph. Overall, we need Sort($|V|$) I/O operations. The semi-external algorithm iterates through all nodes of graph G and updates the cluster IDs of each node. This can be done using $\mathcal{O}(1)$ I/Os. If the number of nodes of the quotient graphs decreases geometrically then uncoarsening of the complete hierarchy can be done using Sort($|V|$) I/O operations. After each solution transfer step, we apply a size-constrained LPA (using $L_{\max}$) to improve the solution in (semi-)external memory on the current level.

References

This chapter is based on Akhremtsev, Sanders and Schulz [3].

Chapter 11

Hypergraph Partitioning

Hypergraph partitioning (HGP) is an important problem with many application areas. Two prominent areas are VLSI design and scientific computing (e.g. the acceleration of sparse matrix-vector multiplications) [105]. While the former is an example of a field where small optimizations can lead to significant savings, the latter is an example where hypergraph-based modeling better captures the objectives of the application domain [31] than graph-based approaches. We focus on a version of the problem that partitions the vertices of a given hypergraph into k blocks of roughly equal size (in our case $1 + \varepsilon$ times the average block size) while optimizing an objective function. In this chapter, we minimize the total cut size, i.e., the number of hyperedges that span multiple blocks. This chapter is mostly based on [64].

Preliminaries

An *undirected hypergraph* $H = (V, E, c, \omega)$ is defined as a set of vertices V and a set of hyperedges E with vertex weights $c : V \to \mathbb{R}_{\geq 0}$ and hyperedge weights $\omega : E \to \mathbb{R}_{>0}$, where each hyperedge is a subset of the vertex set V (i.e., $e \subseteq V$). In HGP literature, hyperedges are also called *nets* and the vertices of a net are called *pins* [31]. We extend c and ω to sets, i.e., $c(U) := \sum_{v \in U} c(v)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. A vertex v is *incident* to a net e if $\{v\} \subseteq e$. $\mathrm{I}(v)$ denotes the set of all incident nets of v. The *degree* of a vertex v is $d(v) := |I(v)|$. Two vertices are *adjacent* if there exists a net e that contains both vertices. The set $\Gamma(v) := \{u \mid \exists\, e \in E : \{v, u\} \subseteq e\}$ denotes of neighbors of v. The *size* $|e|$ of a net e is the number of its pins. Nets of size one are called *single-node* nets.

A *k-way partition* of a hypergraph H is a partition of its vertex set into k *blocks* $\Pi = \{V_1, \ldots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$, $V_i \neq \emptyset$ for $1 \leq i \leq k$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. We use $b[v]$ to refer to the block id of vertex v. We call a

k-way partition Π *ε-balanced* if each block $V_i \in \Pi$ satisfies a *balance constraint*: $\forall i \in \{1..k\} : |V_i| \leq L_{max} := (1+\varepsilon)\lceil \frac{|V|}{k} \rceil$ for some parameter ε. We call a block V_i *overloaded* if $|V_i| > L_{max}$ and *underloaded* if $|V_i| < L_{max}$.

Given a k-way partition Π, the number of pins of a net e in block V_i is defined as $\Phi(e, V_i) := |\{v \in V_i \mid v \in e\}|$. Net e *is connected* to block V_i, if $\Phi(e, V_i) > 0$. Similarly, a block V_i is *adjacent* to a vertex $v \notin V_i$ if $\exists\, e \in I(v) : \Phi(e, V_i) > 0$. $\mathrm{R}(v)$ denotes the set of all blocks adjacent to v. For each net e, $\Lambda(e) := \{V_i \mid \Phi(e, V_i) > 0\}$ denotes the *connectivity set* of e. The *connectivity* of a net e is the cardinality of its connectivity set: $\lambda(e) := |\Lambda(e)|$ [31]. We call a net *internal* if $\lambda(e) = 1$ and *cut* net otherwise (i.e., $\lambda(e) > 1$). Analogously a vertex that is contained in at least one cut net is called *border vertex*.

The *k-way hypergraph partitioning problem* is to find an ε-balanced k-way partition of a hypergraph H that minimizes the *total cut* $\omega(E')$ where $E' := \{e \in E : \lambda(e) > 1\}$ for some ε. This problem is known to be NP-hard [90].

Contracting a pair of vertices (u, v) means merging v into u. The weight of u becomes $c(u) := c(u) + c(v)$ and we connect u to the former neighbors $\Gamma(v)$ of v. We refer to u as the *representative* and v as the *contraction partner*. This process can lead to parallel nets (i.e., $\exists\, e_i, e_j \in E : e_i \triangle e_j \neq \emptyset$, where $\triangle$ is the symmetric difference). In this case, we choose net e_i as representative, update its weight to $\omega(e_i) = \omega(e_i) + \omega(e_j)$ and remove e_j from H. If a contraction creates single-node nets we remove them from the hypergraph, since such nets can never become part of the cut. *Uncontracting* a vertex u undoes the contraction and restores removed parallel and single-node nets. The uncontracted vertex v is put in the same block as u and the weight of u is set back to $c(u) := c(u) - c(v)$. Most definitions about regular graphs (such as weight functions) can be extended to hypergraphs canonically.

Representation. Much like a regular graph a hypergraph can be represented visually. A hypergraph with ten vertices and five edges is shown in Figure 11.1. Alternatively hyperedges are split into pins and labels, each pin connects a vertex to a label, with labels having multiple pins. A label with its pins is also referred to as a net (i.e. "net" is synonymous to hyperedge). This introduces another representation of the hypergraph using vertices, pins and labels. Figure 11.2 shows the same hypergraph as 11.1 with edges as pins and labels (marked with letters for identification) instead of colored sets.

Vertices and the labels of a hypergraph can be interpreted as the two sets of vertices of a regular bipartite graph with the pins connecting them as edges. This introduces a third presentation of the hypergraph shown in Figure 11.3.

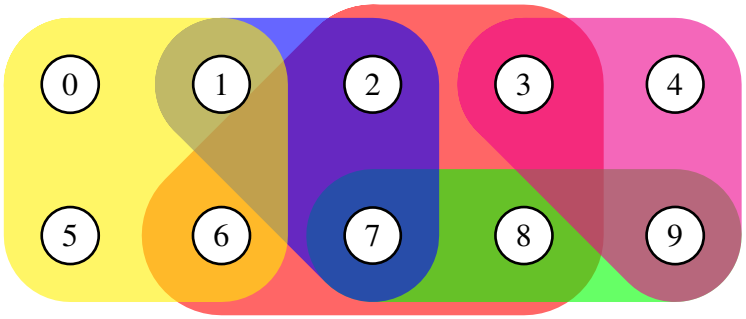


Figure 11.1: A hypergraph with ten vertices and five differently coloured edges

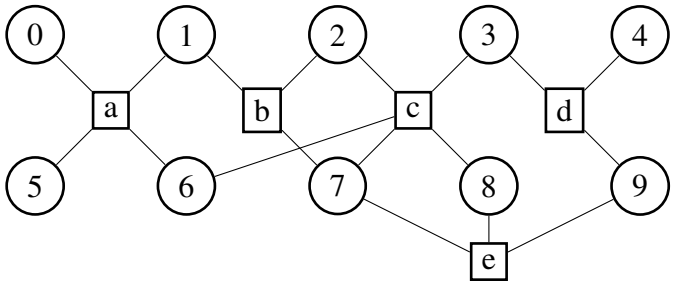


Figure 11.2: The same hypergraph as in Figure 11.1 with pins and labels instead of coloured sets.

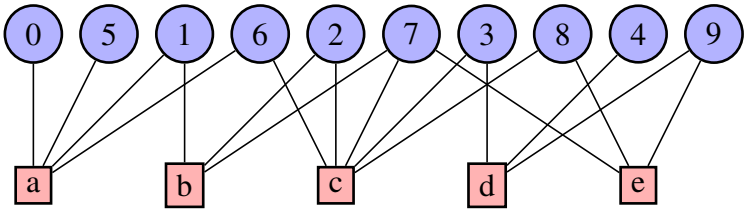


Figure 11.3: A hypergraph as a bipartite graph with the vertices (blue) in one partition and the labels (red) in the other.

11.1 n-Level k-way Hypergraph Partitioning

We now present the main contributions of [64]. As other multilevel algorithms our n-level hypergraph partitioning framework has a coarsening, initial partitioning and an uncoarsening phase. During the coarsening phase, we successively shrink the hypergraph by contracting only *a single pair* of vertices *at each level*, until it is small enough to be initially partitioned by some other partitioning algorithm. We describe the details of our coarsening algorithm in Section 11.1.1 and briefly discuss initial partitioning in Section 11.1.2. The initial solution is transferred to the next finer level by performing a *single* uncontraction step. Afterwards, one of our localized local search algorithms described in Section 11.1.3 and Section 11.1.4 is used to further improve the solution quality.

11.1.1 Coarsening

The vertex pairs (u, v) to be contracted are chosen according to a rating function. The goal of the coarsening phase is to contract highly connected vertices such that the number of nets remaining in the hypergraph and their size is successively reduced [75]. Removing nets leads to simpler instances for initial partitioning, while small net sizes allow FM-based local search algorithms to identify moves that improve the solution quality. Our coarsening algorithm therefore prefers vertex pairs that have a large number of heavy nets with small size in common. This score is then inversely scaled with the product of the vertex weights $c(v)$ and $c(u)$ to keep the vertex weights of the coarse hypergraph reasonably uniform:

$$r(u,v) := \frac{1}{c(v) \cdot c(u)} \sum_{e \in \{I(v) \cap I(u)\}} \frac{\omega(e)}{|e| - 1}. \tag{11.1}$$

This scaling factor was already effective in n-level graph partitioning [102]. At the beginning of the coarsening algorithm, all vertices are rated in random order, i.e., for each vertex u we compute the ratings of all neighbors $\Gamma(u)$ and choose the vertex v with the highest rating as contraction partner for u. Ties are broken randomly. For each vertex, we insert the vertex pair with the highest score into an addressable PQ using the rating score as key. This allows us to efficiently choose the next vertex pair that should be contracted. After contraction, we remove v from the PQ. We then remove all parallel- and single-node nets in $I(u)$. The latter are easily identified, because $|e| = 1$. For parallel hyperedge detection we use an efficient algorithm similar to the one in [63], which is used to identify vertices with identical structure in a graph: We create a fingerprint for each net e : $f_i := \bigoplus_{v \in e} v \oplus x$, for some seed x. These fingerprints are then sorted, which brings potentially parallel nets together. A final scan over the fingerprints then

identifies parallel nets: Only if two consecutive fingerprints f_i, f_j are identical, we have to check whether $e_i \triangle e_j = \emptyset$ by comparing their pins.

Since each contraction potentially influences the rating scores of all neighbors $\Gamma(u)$, we have to recalculate their ratings and update the PQ accordingly. To avoid unbalanced inputs for the initial partitioning phase, vertices v with $c(v) > c_{max} := s \cdot \frac{c(V)}{t}$ are never allowed to participate in a contraction step and are thus removed from the PQ. We refer to this algorithm as *full*.

As in the graph partitioning case, the n-level approach has the advantage that it obviates the need to employ a matching or clustering algorithm to determine the vertices to be contracted. However, this comes at the expense of continuously re-rating the neighbors $\Gamma(u)$ adjacent to the representative. In hypergraph partitioning, this is the most expensive part of the algorithm, because after each contraction, we have to look at all pins of all incident nets $I(v)$. The re-rating can therefore easily become the bottleneck – especially if H contains large nets. To improve the running time of the coarsening phase in these cases, we developed two variations of the full algorithm. Both variations only differ in the way the re-rating of adjacent vertices is handled. After contracting the vertex pair (u, v), the first version only updates the rating of those neighbors, which had chosen either the representative u or the contracted vertex v as contraction partner. This can be done efficiently by maintaining the set $L_w := \{u \mid (u, w) \in \text{PQ}\}$ of all representatives that choose w as contraction partner. The re-rating step then only reevaluates the rating function for each vertex in $L_u \cup L_v$. All other ratings are left untouched. We refer to this version as *partial*. The second variation does not re-rate any vertices immediately after the contraction. Instead, all adjacent vertices $\Gamma(u)$ are marked as *invalid*. If the PQ returns an invalid vertex, we recalculate its rating and update the priority queue accordingly. In case the queue returns a valid rating, we normally continue with the coarsening process. This version is referred to as *lazy*.

11.1.2 Initial Partitioning

The coarsening process is repeated until the number of remaining vertices is below $160k$ or the priority queue becomes empty. The hypergraph is then small enough to be initially partitioned by an initial partitioning algorithm. We use the recursive bisection variant of hMetis for initial partitioning, because it produces better initial partitions than PaToH. In this variant of hMetis, the maximum allowed imbalance of a partition is defined differently [75]: An imbalance value of 5, for example, allows each block to weigh between $0.45 \cdot c(V)$ and $0.55 \cdot c(V)$ *at each bisection step*. We therefore translate our maximum allowed block weight to match this

definition, i.e., we use imbalance parameter

$$\varepsilon' := 100 \cdot \left(\left(\frac{1+\varepsilon}{k} + \frac{\max_{v \in V} c(v)}{c(V)} \right)^{\frac{1}{\log_2(k)}} - 0.5 \right) \tag{11.2}$$

for initial partitioning with hMetis.

11.1.3 Localized direct k-way FM Local Search

Our local search algorithm follows ideas similar to the k-way FM-algorithm proposed by Sanchis [115] and is further inspired by the local search algorithm used by Sanders and Osipov [102]. Sanchis uses $k(k-1)$ PQs to be able to maintain all possible moves for all vertices. We reduce the number of PQs to k – one queue P_i for each block V_i. In contrast to Sanchis, we only consider to move a vertex to *adjacent* blocks rather than calculating and maintaining gains for moves to *all* blocks. This simultaneously reduces the memory requirements and restricts the search space of the algorithm to moves that are more likely to improve the solution. Another key difference is the way a local search pass is started: Instead of initializing the PQs with all vertices or all border vertices, we perform a highly localized local search starting only with the representative and the just uncontracted vertex. The search then gradually expands around this vertex pair by successively inserting moves for neighboring vertices into the queues.

Algorithm Outline. At the beginning of a local search pass, all queues are empty and disabled. A disabled PQ will not be considered when searching for the next move with the highest gain. All vertices are labeled inactive and unmarked. Only unmarked vertices are allowed to become active. To start the local search phase after each uncontraction, we activate the representative and the just uncontracted vertex if they are border vertices. Otherwise, no local search phase is started. *Activating* a vertex v means that we calculate the *gain* $g_i(v)$ for moving v to all adjacent blocks $i \in R(v) \setminus \{b[v]\}$ and insert v into the corresponding queues P_i using $g_i(v)$ as key. The gain $g_i(v)$ is defined as:

$$g_i(v) := \sum_{e \in I(v)} \{\omega(e) : \Phi(e,i) = |e| - 1\} - \sum_{e \in I(v)} \{\omega(e) : \lambda(e) = 1\}. \tag{11.3}$$

Thus, instead of considering all $k-1$ possible moves of a vertex v, we only examine moves to those blocks that are in the union of the connectivity sets of its incident nets: $\bigcup_{e \in I(v)} \{\Lambda(e) \setminus b[v]\}$. After insertion, all PQs corresponding to *underloaded* blocks become enabled. Since a move to an overloaded block will never be feasible, all queues corresponding to overloaded blocks are left disabled. The algorithm then

repeatedly queries only the *non-empty, enabled* queues to find the move with the highest gain $g_i(v)$, breaking ties randomly. Vertex v is then moved to block V_i and labeled inactive and marked. Since each vertex is allowed to move at most once during each pass, we remove all other moves of v from the PQs. We then update all neighbors of v and continue local search until either no non-empty, enabled PQ remains or a constant number of c moves neither decreased the cut nor improved the current imbalance. The latter criterion is necessary, because otherwise the n-level approach could lead to $|V|^2$ local search steps in total. After local search is stopped, we undo all moves until we arrive at the lowest cut state reached during the search that fulfills the balance constraint. All vertices become unmarked and inactive and the algorithm is then repeated until no further improvement is achieved.

Activation and Gain Computation. For each vertex v, we are only interested in gain values for moves to adjacent blocks $R(v)$. Calculating these gains can be done efficiently by looking at all incident nets $I(v)$ and all adjacent blocks *exactly once*. $R(v)$ can be generated on-the-fly while iterating over $I(v)$. To calculate the gain, we distinguish two cases for each net e: If $\lambda(e) = 1$ and $|e| > 1$, net e is internal in block $b[v]$ and will become a cut net when v is moved to another block. If $\lambda(e) = 2$ and $\Phi(e, V_i) = |e| - 1$ for one block V_i of the two blocks in $\Lambda(e)$, net e can be removed from the cut by moving v to block V_i. All other nets cannot be removed from the cut by moving v. Let ω_{int} be the sum of the weights of all internal nets and $\Omega[V_i]$ be the weight of all nets that could be removed from the cut by moving v to block V_i. By iterating over $R(v)$, we can compute the gain values for moving v to all connected blocks V_i: $g_i(v) = \Omega[V_i] - \omega_{int}$.

Update of Neighbors. After moving a vertex v from block V_{from} to a different block V_{to}, we have to update all of its neighbors $\Gamma(v)$. All previously inactive neighbors are activated as described above. All neighbors that became internal are labeled inactive and all corresponding moves are deleted from the PQs. Finally, we update the gains for all moves of the remaining neighbors that are already active and remain border vertices. We reuse the gain values that are already calculated and only perform *delta-gain-updates*: If the move changed the contribution to the gain values for a net $e \in I(v)$, we account for that change by incrementing/decrementing the gains of the corresponding moves by $\omega(e)$.

For each active vertex, the PQs maintain the gains to all *adjacent* blocks. The set of adjacent blocks, however, is subject to change during local search, because vertex movements can increase as well as decrease the connectivity of incident nets. The update process therefore has to take these changes into account. Otherwise we would either miss potential moves or perform *stale* moves, i.e., move a vertex to a block that was adjacent to v at some point of the local search but is not adjacent

any more at the time it is returned by the PQ. If the move of v inserted V_{to} into $R(u)$ for a neighbor $u \in \Gamma(v)$, we calculate the gain $g_{\text{to}}(u)$ and insert u into queue P_{to}. Similarly, if the move removed V_{from} from $R(u)$, we remove the vertex u from P_{from}.

Critical and Locked Nets. Performing delta-gain-updates rather than recalculating the gains for each affected vertex from scratch considerably reduces the complexity of the update step. We employ two additional techniques to further speed-up the gain update: (i) We generalize the notion of *critical nets* introduced by Fiduccia and Mattheyses [46] for bipartitioning to k-way partitioning. (ii) We exclude nets from the gain-update process that can not change their gain contribution, because the net is *locked* in two blocks and therefore cannot be removed from the cut in the current local search pass. The concept of *locked nets* was first described by Krishnamurthy [85] and transferred to k-way partitioning by Sanchis [115].

11.1.4 Size-Constrained Label Propagation in HGP

The *label propagation algorithm* for graph clustering was recently equipped with a size constraint to work as a coarsening and a local search algorithm for graph partitioning [95]. We briefly outline the previous local search algorithm before describing our adaptation to hypergraphs. Initially, each vertex v is in its current block $b[v]$. The algorithm then works in rounds. In each round, the vertices are visited in random order and each vertex v is moved to the eligible (i.e. not overloaded after the move) block V_i that has the strongest connection to v. Ties are broken randomly. After all vertices are visited, the process is repeated until the labels have converged or a maximum number of ℓ rounds is reached.

We modify this local search algorithm as follows in order to be applicable in our n-level hypergraph partitioning context. Instead of iterating over all vertices, we start the first iteration only with the vertex pair (u, v) that has just been uncontracted. If one of these two vertices changes its block, all of its neighbors are allowed to change their block in the next iteration. This can be done efficiently by maintaining two queues Q_1 and Q_2. Q_1 contains the vertices for the current iteration and Q_2 those for the next iteration. After each round, we clear Q_1 and swap it with Q_2. In order to reflect our partitioning objective, we move a vertex to the eligible block that maximizes the *gain* as defined in Equation 11.3. Finally, we adapt the tie-breaking scheme to successively reduce the number of blocks a net is connected to: If multiple blocks have the same maximum gain, we choose to move the vertex to the block that leads to the highest connectivity decrease for all incident nets. Only in case multiple blocks also have the same connectivity decrease value, we resort to random tie-breaking.

References

This chapter is largely based on Bichot and Siarry [19], and Henne, Meyerhenke, Sanders, Schlag, and Schulz [64]. This chapter was created by Jan Jacob and Christian Schulz.

Chapter 12

Graph Clustering

In this chapter, we cover algorithms to compute a clustering of a graph. The first method is greedy agglomeration, which uses global information to compute a clustering. We then look at the Louvain method which uses local movements and the multilevel idea to improve modularity instead. We finish this chapter with minimum cut tree clustering which is able to give performance guarantees.

12.1 Greedy Agglomeration

The first algorithm that we look at is greedy agglomeration. Roughly speaking, the algorithm starts with a singleton clustering and iteratively merges/joins clusters. In the end, the best intermediate clustering is chosen. To describe the network's community structure, a dendrogram is used. The hierarchical decomposition of the network is presented at all levels. Thus, the dendrogram is a tree whose leaves are the vertices of the original network and whose internal nodes correspond to the joins. To proceed, we recall the modularity:

$$Q \;=\; \frac{1}{2m} \sum_{v,w \in V} \Big[A_{vw} - \frac{k_v k_w}{2m} \Big] \delta(c_v, c_w)$$

where the function $\delta(i, j)$ is 1 if $i = j$ and 0 otherwise, and A_{vw} denotes the entry of adjacency matrix indexing vertices v and w. The community of vertex v is denoted by c_v and the vertex degree is k_v. At each iteration of this method, pairs of communities are joined and the modularity is subsequently calculated. More precisely, in each iteration the pair of communities yielding the highest gain in modularity is merged. The most naive implementation of this process involves storing the adjacency matrix of the graph as an array of integers and repeatedly merging pairs of rows and columns. Clearly this is inefficient. Before describing a more

efficient variant, we define two quantities: $e_{ij} = \frac{1}{2m}\sum_{v,w\in V} A_{vw}\delta(c_v,i)\delta(c_w,j)$ which is the fraction of edges that join vertices in community i to vertices in community j, with $\delta(c_v,c_w) = \sum_i \delta(c_v,i)\delta(c_w,i)$, and $a_i = \frac{1}{2m}\sum_{v\in V} k_v\delta(c_v,i)$ which is the fraction of node ends that are attached to community i. We can rewrite the modularity formula:

$$\begin{aligned}
Q =& \frac{1}{2m}\sum_{v,w\in V}\Big[A_{vw} - \frac{k_v k_w}{2m}\Big]\sum_i \delta(c_v,i)\delta(c_w,i) \\
=& \sum_i \Big[\frac{1}{2m}\sum_{v,w\in V} A_{vw}\delta(c_v,i)\delta(c_w,i) - \frac{1}{2m}\sum_{v\in V} k_v\delta(c_v,i)\frac{1}{2m}\sum_{w\in V} k_w\delta(c_w,i)\Big] \\
=& \sum_i (e_{ii} - a_i^2)
\end{aligned}$$

A way to envisage the naive implementation of this process is by thinking of the network as a multigraph, in which a whole community is represented by a vertex, bundles of edges connect vertices and internal community edges are self-edges. Joining communities i and j corresponds to replacing the ith and jth rows and columns by their sum. Calculating ΔQ_{ij} and finding the pair i, j with largest ΔQ_{ij} is time consuming and efficient solutions can be accomplished for sparse graphs.

An efficient method is described in [1]. The adjacency matrix is not maintained and ΔQ_{ij} is not repeatedly calculated. Differently, a matrix of value of ΔQ_{ij} is maintained and those values are updated. The key idea for that is summarized by the following theorem, which states that in case there are no edges between communities, the value of Q never increases.

Theorem 31
Joining a pair of communities between which there are no edges can never result in an increase in Q.

Proof. We consider the modularity value before and after merging two clusters. By definition, $Q_{before} = (e_{AA} - a_A^2) + (e_{BB} - a_B^2) = (e_{AA} + e_{BB}) - (a_A^2 + a_B^2)$. By definition of merging step, $V(C) = V(A)\cup V(B)$. This implies $a_C = a_A + a_B$, because $\delta(c_v, C) = 1$ iff $\delta(c_v, A)\delta(c_v, B) = 1$ for any $v \in V(A)\cup V(B)$. From our hypothesis, $E(C) = E(A)\cup E(B)$ with $E(A)\cap E(B) = \emptyset$. This implies $e_{CC} = e_{AA} + e_{BB}$, because no edge is removed or added. Thus, $Q_{after} = (e_{AA} + e_{BB}) - (a_A + a_B)^2$, but $(a_A + a_B)^2 \geq a_A^2 + a_B^2$, so $Q_{after} \leq Q_{before}$. □

In this way, we observe that non-adjacent clusters don't yield larger Q values and don't need to be checked during the iteration. However, adjacent clusters do, and the following theorem explains how their value changes within an iteration.

Theorem 32
The change in modularity upon joining two communities A and B is given by $\Delta Q = e_{AB} + e_{BA} - 2a_A a_B$.

Proof. We consider the modularity value before and after merging two clusters. Let $Q_{before} = (e_{AA} + e_{BB}) - (a_A^2 + a_B^2)$ and $Q_{after} = (e_{CC} + (e_{AB} + e_{BA})) - a_C^2$, where E' is the edge set connecting the adjacent clusters with $|E'| = e_{AB} + e_{BA}$. We know that $a_C = a_A + a_B$, because $V(C) = V(A) \cup V(B)$, so $a_C^2 = a_A^2 + 2a_A a_B + a_B^2$. On the other side, $E(C) = E(A) \cup E(B) \cup E'$, because merges of adjacent clusters are being considered, so $e_{CC} = e_{AA} + e_{BB}$. The modularity change is given by $\Delta Q = Q_{after} - Q_{before} = e_{AB} + e_{BA} - 2a_A a_B$. □

Due sparseness, we can represent the matrix of values ΔQ_{ij} row-by-row, where each row is a balanced binary tree, so that inserts and lookups are done in $\mathcal{O}(\log n)$ time, combined with a deduplication of this information in a max-heap, so that *find-min* works in $\mathcal{O}(1)$ time. In order to identify i, j corresponding to the largest ΔQ_{ij} in $\mathcal{O}(1)$ time at each iteration, we keep the maximum element of each row in max-heap. As of the last theorem, we need the values a_i to update the ΔQ_{ij} values and those are stored in a simple array.

Theorem 33
If community l is connected to both k and j, then $\Delta Q'_{jl} = \Delta Q_{kl} + \Delta Q_{jl}$
If community l is connected to both k but not j, then $\Delta Q'_{jl} = \Delta Q_{kl} - 2a_j a_l$
If community l is connected to both j but not k, then $\Delta Q'_{jl} = \Delta Q_{kl} - 2a_k a_l$

Note that this implies a single peak over the course of the algorithm: if $\Delta Q < 0$, all the ΔQ can only decrease.

Algorithm. An initialization step is carried on prior to algorithm execution as it follows. (1) start with singletons, (2) set $e_{kj} = 1/2m$ if k and j are connected, zero otherwise, (3) set $a_i = k_i/2m$ and (4) finally $\Delta Q_{kj} = (1/2m) - (k_k k_j)/(4m^2)$ if k, j are connected and zero otherwise. The following steps are performed after initialization:

1. select largest ΔQ_{kj} and populate max-heap with in-row largest elements
2. update matrix, max-heap and a_i
3. increment Q
4. repeat 2-4 until only a single community remains

Analysis. We use the $deg(i)$ notation also to describe the number of neighboring communities in the reduced graph during agglomeration. Firstly, we analyze the algorithm join step, whose most expensive step is to update the state. The jth row is firstly updated by summing up the contents of row i, i.e. there are deg(i) items to be inserted into the balanced binary tree representing the jth row and it takes $\mathcal{O}(\log deg(i)) \leq \mathcal{O}(\log n)$. The remaining elements of jth row are updated, which are at most $deg(i) + deg(j)$. In the kth row, we update a single element what costs $\mathcal{O}(\log deg(k)) \leq \mathcal{O}(\log n)$. In this way, updating the row costs a total of $\mathcal{O}((deg(i) + deg(j)) \log n)$ time. The max-heaps must be also actualized. Reforming the max-heap for the jth row costs $\mathcal{O}(deg(j))$ time. For the kth row, $\mathcal{O}(\log k) \leq \mathcal{O}(\log n)$ time is required, in order to insert, raise or lower ΔQ_{kj}. Updating the values $a'_j = a_j + a_i$ can be done in $\mathcal{O}(1)$ time. The worst-case assumption is that the degree of a community is the sum of the degrees of all the vertices in the original network comprising it. In that case, each vertex of the original network contributes its degree to all of the communities it is a part of. If the dendrogram has depth d, there are at most d nodes in this path, and since the total degree of all the vertices is $2m$, we have a running time of $\mathcal{O}(md \log n)$ time.

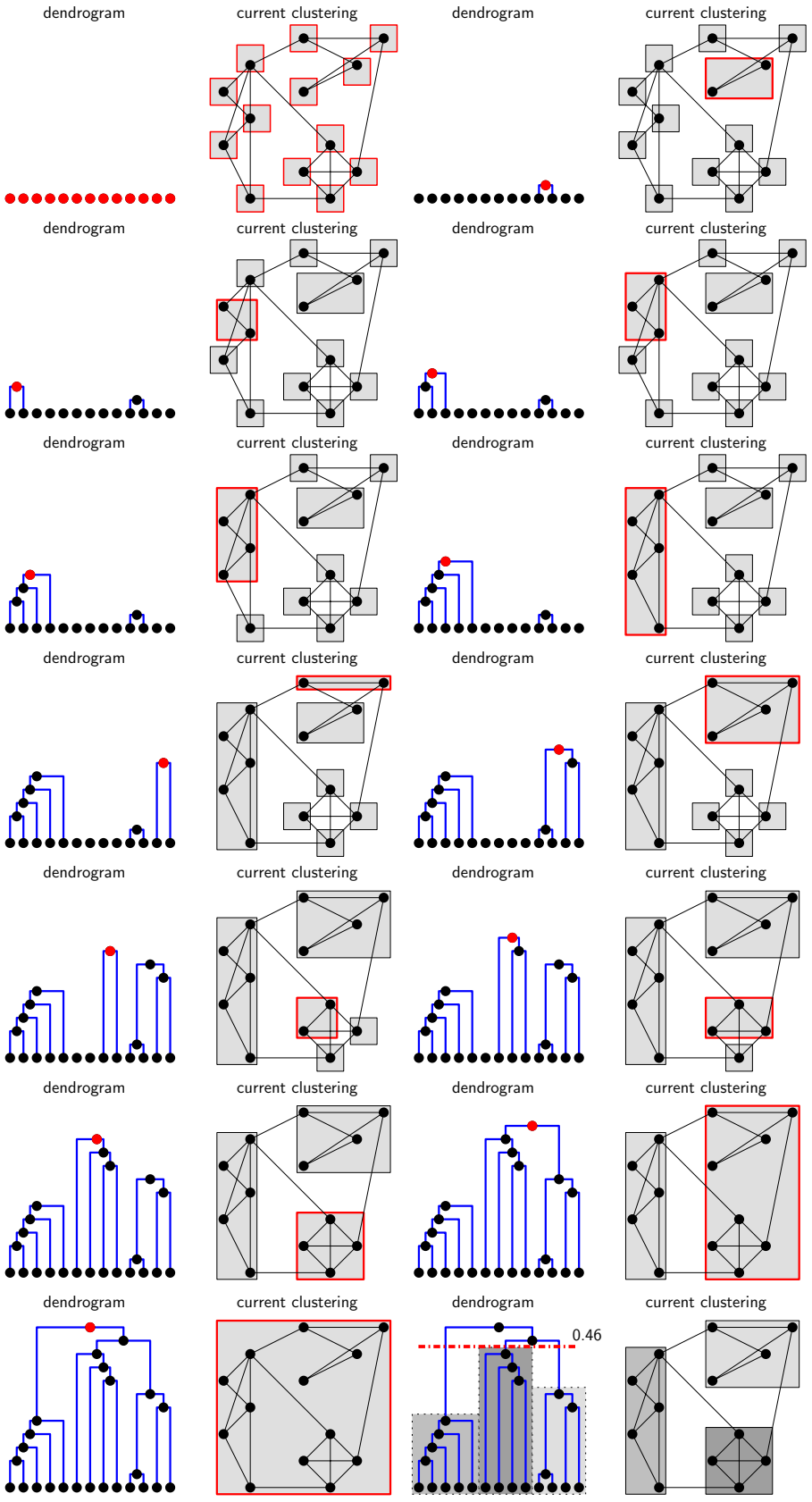


Figure 12.1: An example run of greedy agglomeration.

12.2 Louvain Method

This is a simple heuristic method which is based on modularity optimization and works fast. The basic techniques involved are the *local movement* and *multi-level clustering*. At the beginning, each node is a singleton cluster, then the nodes are traversed in random order and move to the neighboring cluster yielding the highest modularity increase. A single step of the local movement is illustrated in the Figure 12.2.

Algorithm. The Louvain method is quite simple, therefore easier to implement (e.g. for the absence of heaps). The algorithm runs in two phases. In the *first phase*, the algorithm is firstly initialized, so that each node is assigned to a different community. Then, we iteratively proceed with following steps, targeting to find local maxima:

1. pick a node u at random
2. remove u from cluster, compute removal gain ΔQ
3. for each cluster in $N(u)$ compute add gain $\Delta Q'$
4. if highest total gain is positive ($\Delta Q'$ - $\Delta Q > 0$), then move node

This first phase stops when a local maxima of the modularity is attained, that is when no node move yields modularity gain. The modularity gain by moving an isolated node u into a community C can be computed by

$$\Delta Q = \frac{s}{m} + \frac{deg(u)}{4m^2} + \frac{(\sum_{v \in C} deg(v))^2}{4m^2} - \frac{(deg(u) + \sum_{v \in C} deg(v))^2}{4m^2}$$

where s is the number of edges that u has pointing inside C. A similar expression is used in order to evaluate the change of modularity when i is removed from its community. Observe that the algorithm's output depends on the order in which

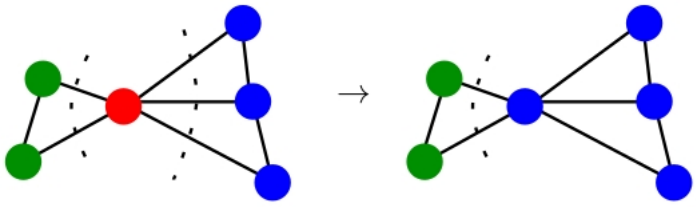


Figure 12.2: Local movement in Louvain method.

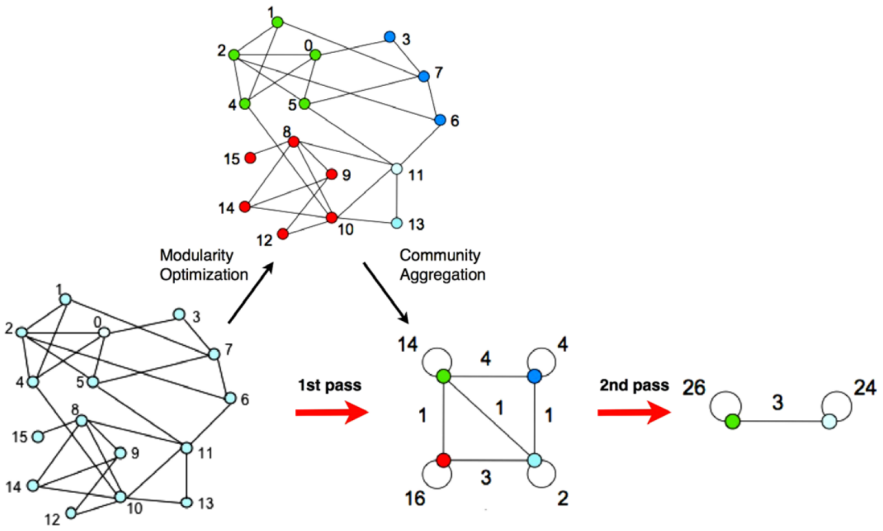


Figure 12.3: Schematic execution of the Louvain method.

the nodes are processed. Thus, this is an important aspect for performance improvement. The *second phase* of the algorithm consists in building a new network whose nodes are now the communities found during the first phase. This is done by adjusting the node and edge weights, so that a subsequent first phase can be performed on the new graph. The weights of the links between the new nodes are given by the sum of the weight of the links between nodes in the corresponding two communities. The combination of both phases is called a *pass*.

Figure 12.3 illustrates the mechanics of the Louvain method. On the left-side down the original input graph is shown, which is optimized according to the modularity. Afterwards, each community is synthesized to a single node and edges between the adjacent communities are created with the summed weights. The second pass is a repetition of this procedure, which ends because no further improvement can be done.

Discussion. The involved steps are intuitive and easy to implement. Since the modularity gain formula is easy to compute, the algorithm also runs very fast. The first phase is the most time-consuming part of the algorithm and in this phase, communities are formed. Thus, the number of meta-communities decreases at each pass and the result is that subsequent passes turn out to be always faster. In comparison to the greedy agglomeration method, Louvain method works by moving nodes, rather than merging intermediary clusters.

12.3 Minimum Cut Tree Clustering

This approach is based on maximum flow techniques, and in particular minimum cut trees [50]. A single assumption is done, that the graph is connected. The main idea is to create clusters that have small inter-cluster cuts and relatively large intra-cluster cuts. An artificial sink is inserted into the graph, which gets connected to all nodes in the network. Maximum flows are calculated between all nodes of the network and the artificial sink, yielding a min-cut tree constructed via $(n-1)$ max-flows. For every undirected graph G, there exists a corresponding weighted graph T_G, called the minimum cut tree of G. The min-cut tree has the property that we can find the minimum cut between two nodes s, t by inspecting the path connecting both nodes and looking for the edge with smallest weight. The capacity of the edge is equal to the min-cut value and its removal yields two node sets in T_G.

If $G(V, E)$ is an undirected graph, and $s, t \in V$ be two nodes of G, then (S, T) is the minimum cut between s and t, where $s \in S$ and $t \in T$. In this case, S is the community of s with respect to t. If (S, T) is not unique, the minimum cut maximizing the size of S is chosen and this makes S unique.

The definition of community S is extended, when no t is given. For that, an artificial node t is introduced, which we call *artificial sink*. This special node is connected to all other nodes with edge capacity equal to α, a parameter to be chosen. The following theorem expresses the bound on this value.

Theorem 34
Let $G(V, E)$ be an undirected graph, $s \in V$ a source, and connect an artificial sink t with edges of capacity α to all nodes. Let S be the community of s with respect to t. For an non-empty P and Q, such that $P \cup Q = S$ and $P \cap Q = \emptyset$, the following bound always hold:

$$\underbrace{\frac{c(S, V-S)}{|V-S|}}_{\substack{\textit{intra-cluster} \\ \textit{expansion}}} \leq \alpha \leq \underbrace{\frac{c(P, Q)}{min(|P|, |Q|)}}_{\substack{\textit{inter-cluster} \\ \textit{expansion}}}$$

A vertex v that served to identify a cluster-defining v-t-cut is called the representative of the respective cluster. The parameter α gives a quality guarantee and scaling it yields a nested hierarchy of clusterings. Putting the pieces together, we can summarize this as a procedure:

1. start with the original graph G
2. put the artificial sink t and its corresponding edges with weight α

3. calculate the min-cut tree
4. delete t and the added edges

At step 4, the graph induced by the deletion of the artificial node and the added edges is a set of connected components, which form the clustering. Figure 12.4 illustrates the min-cut tree clustering algorithm.

Discussion. While flows and cuts are well-defined for both directed and undirected graphs, minimum cut trees are defined only for undirected graphs. However, this is not a restriction of the method, because there is a way to deal with it [50]. The algorithm has high runtime, namely $\mathcal{O}(n)$ max-flow computations are required. Furthermore, the user needs to choose a suitable α carefully.

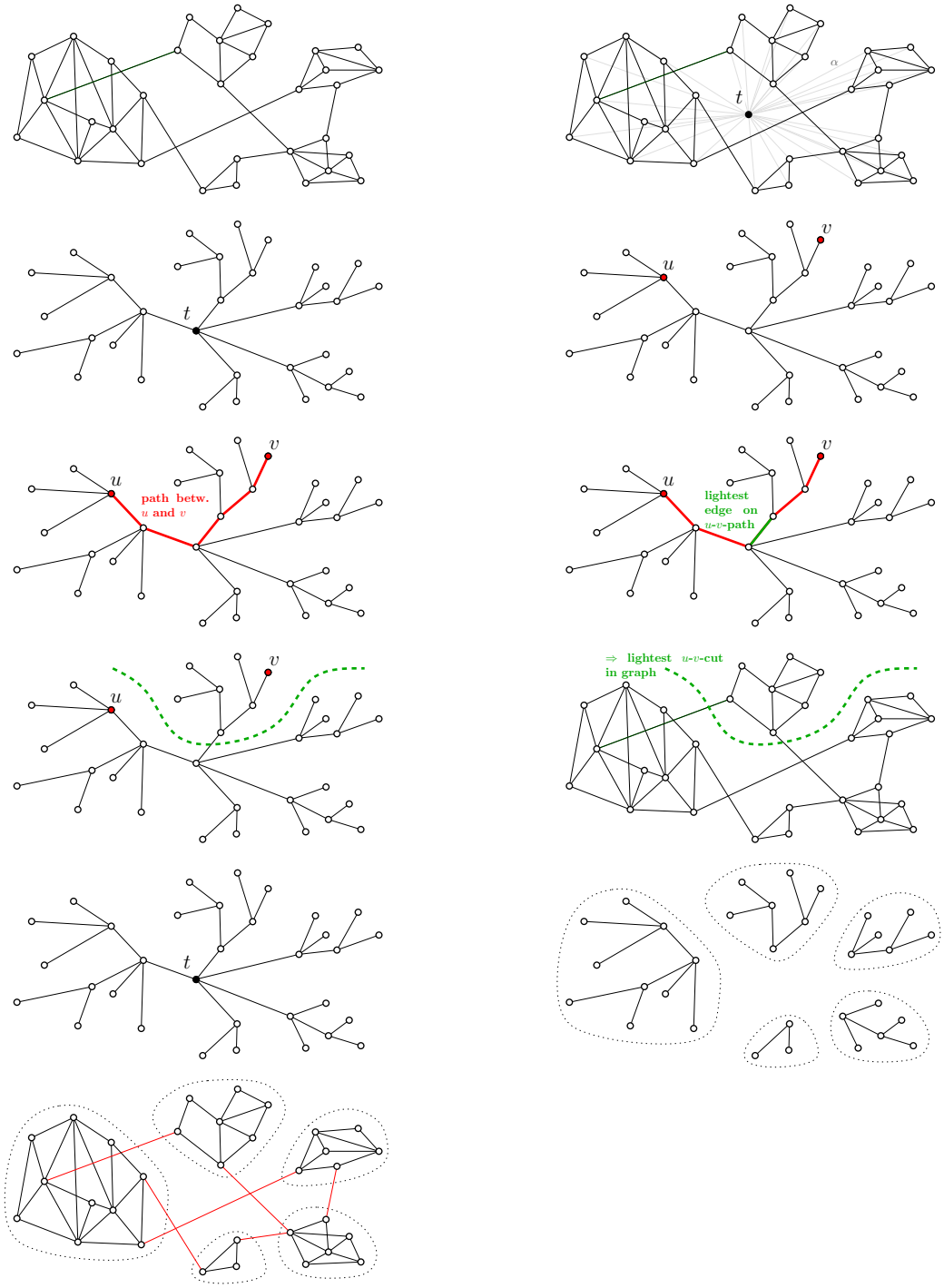


Figure 12.4: Illustration of Min-Cut Tree Clustering.

12.4 Dynamic Clustering

Often we need to face the following problem. We are given a network G and a clustering C for that, want to change the network into G' and obtain the corresponding clustering C' without the need to recalculate the clustering. Figure 12.5 illustrates this. In dynamic instances, network changes involve the group structure, and the dynamic approach is to update the previous clustering, while reacting to changes in the graph. The criteria are that changes can be propagated fast, while preserving clustering quality with smooth transitions. Heuristics based on locality offer no provable quality, since it assumes that local changes have local consequences. In this approach, changes in the graph invalidate only the affected clusters. Hereby, small changes imply on smooth transitions, as a small search space (only affected parts are considered) implies on fastness. However, due local optimization, quality is an open question.

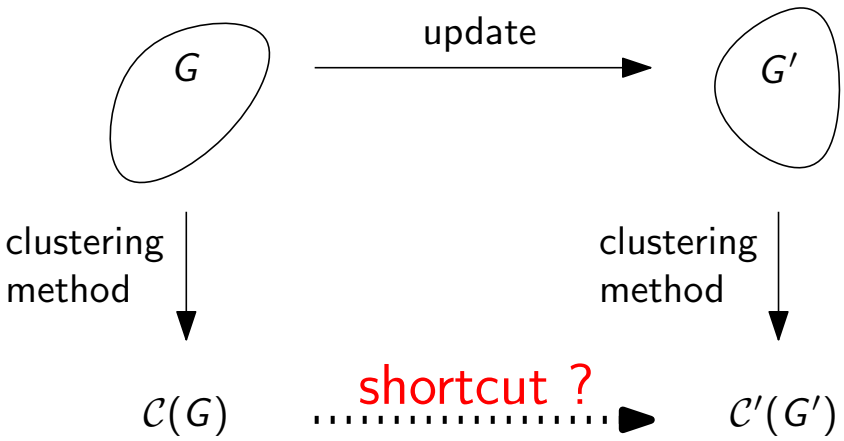


Figure 12.5: Dynamic clustering problem.

References

This chapter is largely based on Buluc et al. [29], Delling et al. [38], Görke [59] and Schulz [122]. This chapter was created by Christian Schulz.

Chapter 13

Evolutionary Algorithms

For a general overview of genetic/evolutionary algorithms tackling the graph partitioning problem, we refer the reader to the overview paper by Kim et al. [83]. In this section we focus on the description of hybrid evolutionary approaches that combine evolutionary ideas with the multilevel graph partitioning framework [129, 12, 13]. Other approaches such as Probe by Chardaire et al. [32], which can be viewed as a genetic algorithm without selection, and Fusion Fission by Bichot et al. [18], which is inspired by nuclear processes, are not covered here. Hybrid algorithms are usually able to compute partitions with considerably better quality than those that can be found by using a single execution of a multilevel algorithm.

We start the chapter by giving a general overview to evolutionary algorithms, continue with three evolutionary approaches for the graph partitioning problem and then finish with meta-heuristics for the graph clustering problem.

13.1 Introduction

The general idea behind evolutionary algorithms is to use mechanisms which are highly inspired by biological evolution such as selection, mutation, recombination and survival of the fittest. An evolutionary algorithm starts with a population of individuals and evolves the population into different populations over several rounds. In each round, the evolutionary algorithm uses a selection rule based on the fitness of the individuals of the population to select good individuals and combine them to obtain improved offspring [57]. There are many rules for the selection of individuals from the population. One possibility that has proven to be effective is the tournament selection rule by Miller et al. [97]. Here, the fittest out of two distinct random individuals from the population is selected.

When an offspring is generated, an eviction rule is used to select a member of the population and replace it with the new offspring. In general one has to take both

into consideration, the fitness of an individual and the distance between individuals in the population [10]. There are multiple possibilities to generate offsprings during the course of one generation. One possibility is to only generate one offspring per generation. Such an evolutionary algorithm is called *steady-state* [37]. A typical structure of an evolutionary algorithm is depicted in Algorithm 14. Evolutionary approaches combined with local search heuristics are called meme tic algorithms.

For an evolutionary algorithm it is of major importance to keep the diversity in the population high [10], i.e. the individuals should not become too similar, in order to avoid a premature convergence of the algorithm. In other words, to avoid getting stuck in local optima, a procedure is needed that randomly perturbs the individuals. In classical evolutionary algorithms, this is done using a mutation operator. It is also important to have operators that introduce unexplored search space to the population. Interestingly, Inayoshi et al. [71] noticed that good local solutions of the graph partitioning problem tend to be close to one another. Boese et al. [21] showed that the quality of the local optima overall decreases as the distance from the global optimum increases.

Algorithm 14 A classical general steady-state evolutionary algorithm.

```
procedure steady-state-evolutionary-algorithm
  create initial population P
  while stopping criterion not fulfilled
    select parents p_1, p_2 from P
    combine p_1 with p_2 to create offspring o
    mutate offspring o
    evict individual in population using o
  return the fittest individual that occurred
```

13.2 Evolutionary Algorithms for Partitioning

13.2.1 EA by Edge Weight Perturbation

The first approach that combined evolutionary ideas with a multilevel partitioner was by Soper et al. [129]. The authors define two main operations, a combine and a mutation operation. Both operations modify the edge weights of the graph depending on the input partitions and then use the multilevel partitioner Jostle, which uses the modified edge weights to obtain a new partition of the original graph. The combine operation works as follows (the mutation operation is done in a similar way). The algorithm first computes node biases and then uses those to compute perturbations of the edge weights. However, node biases are not an

input to the multilevel graph partitioner. Given two partitions of the graph, a node is assigned a random value in $[0, 0.01]$ if the node is a boundary node in both input partitions and a *larger* bias of 0.1 plus a random value in the same range, otherwise. For an edge, the perturbed weight is then defined as one plus the biases of its incident nodes. Note that the perturbed edge weights are chosen such that the local search is guided to mimic the input partitions. The algorithm uses a fixed population size of fifty and, to obtain a new generation, creates fifty new individuals using a ratio of 7:3 of combine and mutation operations. A new generation is then defined as the best fifty partitions out of the current generation and the fifty newly created ones. While producing partitions of very high quality, the authors report running times of up to one week. In their paper the authors introduce the well-known Walshaw benchmark, which is presented in the next section. A similar approach based on edge weight perturbations is used by Delling et al. [39]. Note that using perturbations of edge weights changes the structure of the real underlying partitioning problem. Hence, it may take a while for the algorithm to converge.

13.2.2 EA by Core Groups

A multilevel memetic algorithm for the perfectly balanced graph partition problem, i.e. $\epsilon = 0$, was proposed by Benlic et al. [12, 13]. The main idea of their algorithm is that among high quality solutions a large number of nodes will always be grouped together. In their work the partitions represent the individuals. We briefly sketch the combination operator for the case that two partitions are combined. First the algorithm selects two individuals/partitions from the population using a λ-tournament selection rule, i.e. choose λ random individuals from the population and select the best among those if it has not been selected previously. Let the selected partitions be $P_1 = (V_1, \ldots, V_k)$ and $P_2 = (W_1, \ldots, W_k)$. Then sets of nodes that are grouped together within the partitions are computed:

$$B := \left\{ \bigcup_{j=1}^{k} \{V_j \cap W_{\sigma(j)}\} \right\}$$

such that the number of nodes that are grouped together, $\sum_{j=1}^{k} |V_j \cap W_{\sigma(j)}|$, is maximum among all permutations σ of $\{1, \ldots, k\}$. An offspring is created as follows. Nodes in B will be grouped within a block of the offspring. That means if a node is in the set B, then it is assigned to the same block to which it was assigned to in P_1. Otherwise, it is assigned to a random block, such that the balance constraint remains fulfilled. Local search is then used to improve the computed offspring before it is inserted into the population. In Benlic et al. [13] the authors combine their approach with tabu search. Their algorithms produce partitions of

very high quality, but cannot guarantee that the output partition fulfills the desired balance constraint.

13.2.3 EA by Multilevel Combine Operations

We now describe a multilevel combine operator framework [118]. In contrast to previous methods that use a multilevel framework, our combine operators do not need perturbations of edge weights since we integrate the operators into our partitioner and do not use it as a complete black box.

Furthermore, all of our combine operators *assure* that the partition quality of the offspring is *at least as good as the best of both parents*. Roughly speaking, the combine operator framework combines an individual/partition $\mathcal{P} = V_1^{\mathcal{P}}, ..., V_k^{\mathcal{P}}$ (which has to fulfill a balance constraint) with a clustering $\mathcal{C} = V_1^{\mathcal{C}}, ..., V_{k'}^{\mathcal{C}}$. Note that the clustering does not necessarily fulfill a balance constraint and k' is not necessarily given in advance. All instantiations of this framework use a different kind of clustering or partition. The partition and the clustering are both used as input for our multilevel graph partitioner KaFFPa in the following sense. Let $\mathcal{E}$ be the set of edges that are cut edges, i.e. edges that run between two blocks, in either $\mathcal{P}$ *or* $\mathcal{C}$. All edges in $\mathcal{E}$ are blocked during the coarsening phase, i.e. they *are not contracted* during the coarsening phase. In other words, these edges are not eligible for the matching algorithm used during the coarsening phase and therefore are not part of any matching computed. Figure 13.1 illustrates this kind of coarsening.

The stopping criterion of the multilevel partitioner is *modified* such that it stops when no contractable edge is left. Note that the coarsest graph is now exactly the same as the quotient graph $\mathcal{Q}'$ of the overlay clustering of $\mathcal{P}$ and $\mathcal{C}$ of G (see Figure 13.2 gives an example). Hence nodes of the coarsest graph correspond to the connected components of $G_{\mathcal{E}} = (V, E \backslash \mathcal{E})$ and the weight of the edges between nodes corresponds to the sum of the edge weights running between those connected components in G.

As soon as the coarsening phase is stopped, we apply the partition $\mathcal{P}$ to the coarsest graph and use this as initial partitioning. This is possible since we did not contract any cut edge of $\mathcal{P}$. Note that due to the specialized coarsening phase and this specialized initial partitioning, we obtain a high quality initial solution on a very coarse graph which is usually not discovered by conventional partitioning algorithms. Since our local search algorithms guarantee no worsening of the input partition and use random tie breaking, we can assure nondecreasing partition quality. Note that local search algorithms can effectively exchange good parts of the solution on the coarse levels by moving only a few nodes. Figure 13.2 gives an example.

Also note that this combine operator can be extended to be a multi-point combine operator, i.e. the operator would use p instead of two parents. However,

during the course of the algorithm a sequence of two point combine steps is executed which somehow "emulates" a multi-point combine step. Therefore, we restrict ourselves to the case $p = 2$. When the offspring is generated we have to decide which solution should be evicted from the current population. We evict the solution that is *most similar* to the offspring among those individuals in the population that have a cut worse or equal to the cut of the offspring itself. The difference of two individuals is defined as the size of the symmetric difference between their sets of cut edges. This ensures some diversity in the population and hence makes the evolutionary algorithm more effective.

Classical Combine using Tournament Selection

The first instantiation of the combine framework corresponds to a classical evolutionary combine operator C_1. It takes two individuals P_1, P_2 of the population and performs the combine step described above. In this case, $\mathcal{P}$ corresponds to the partition having the smaller cut and $\mathcal{C}$ corresponds to the partition having the larger cut. Random tie breaking is used if both parents have the same cut. The selection process is based on the tournament selection rule by Miller et al. [97], i.e. P_1 is the fittest out of two random individuals R_1, R_2 from the population. The same is done to select P_2. Note that in contrast to previous methods the generated offspring will have a cut smaller than or equal to the cut of $\mathcal{P}$. Due to the fact that our multilevel algorithms are randomized, a combine operation performed twice using the same parents can yield a different offspring. Figure 13.2 illustrates this combine operation.

Cross Combine

In this instantiation of the combine framework C_2, the clustering $\mathcal{C}$ corresponds to a partition of G. But instead of choosing an individual from the population, we create a new individual in the following way. We choose k' uniformly at random

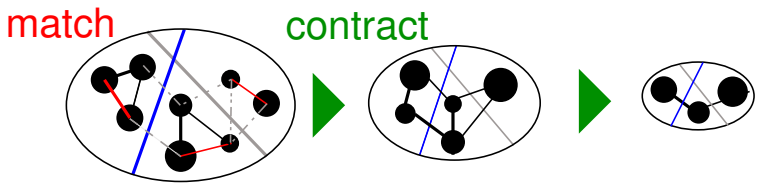


Figure 13.1: A graph G with two partitions, the dark and the light line, are shown. Cut edges are not eligible for the matching algorithm. Contraction is done until no matchable edge is left. The best of the two given partitions is used as initial partition. Source: [118].

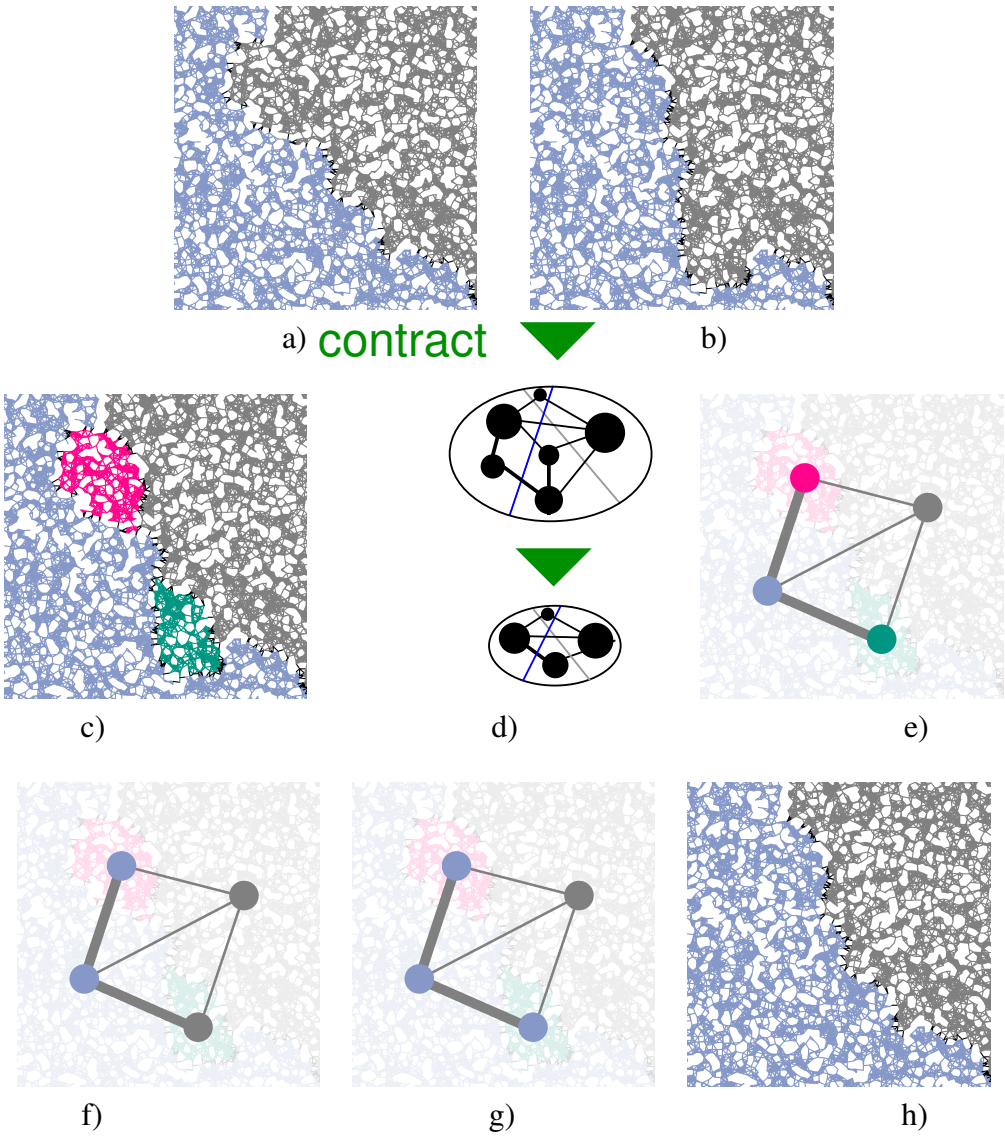


Figure 13.2: An example combine operation of two partitions a), b) of a random geometric graph. In the overlay of these partitions c), only edges that run within the same block can be contracted. The multilevel coarsening phase d) stops as soon as there is no contractable edge left. The resulting graph is the quotient graph of the overlay e). Partition b is applied to this graph f). Local search is applied on all levels of the hierarchy g). We end up with a graph that has the "good" cuts of both input partitions h). Source: [118].

in $[k/4, 4k]$ and ϵ' uniformly at random in $[\epsilon, 4\epsilon]$. We then use KaFFPa to create a k'-partition of G fulfilling the balance constraint $\max_i c(V_i) \leq (1+\epsilon')c(V)/k'$. In general larger imbalances reduce the cut of a partition which then yields good clusterings for our combine operation. To the best of our knowledge there has been no genetic algorithm that performs operations combining individuals from different search spaces. One can extend the idea to combine a partition with an arbitrary graph clustering that fits a specific optimization domain.

Mutation Operators

We define two mutation operators, an ordinary and a modified F-cycle. Both mutation operators use a random individual from the current population. The main idea is to iterate coarsening and uncoarsening several times using different seeds for random tie breaking. The first mutation operator M_1 can assure that the quality of the input partition does not decrease. It is basically an ordinary F-cycle which is an algorithm used in KaFFPa. Edges between blocks are not contracted. The given partition is then used as initial partition of the coarsest graph. In contrast to KaFFPa, we now can use the partition as input to the algorithm in the very beginning. This ensures non-decreasing quality since our local search algorithms guarantee no worsening.

The second mutation operator M_2 works quite similar with the small difference that the input partition is not used as initial partition of the coarsest graph. That means we obtain very good coarse graphs but we cannot assure that the final individual has a higher quality than the input individual. In both cases, the resulting offspring is inserted into the population using the eviction strategy described above.

Putting Things Together and Parallelization

We now explain the parallelization and describe how everything is put together. Each processing element (PE) basically performs the same operations using different random seeds (see Algorithm 15). First we estimate the population size S: each PE performs a partitioning step and measures the time $\bar{t}$ spend for partitioning. We then choose S such that the time for creating S partitions is approximately t_{total}/f where the fraction f is a tuning parameter and t_{total} is the total running time that the evolutionary algorithm is given in advance to produce a partition of the graph. Each PE then builds its own population, i.e. KaFFPa is called several times to create S individuals/partitions. Afterwards the algorithm proceeds in rounds as long as time is left. With corresponding probabilities, mutation or combine operations are performed and the new offspring is inserted into the population.

We choose a parallelization/communication protocol that is quite similar to *randomized rumor spreading* by Doerr et al. [41]. Let p denote the number of

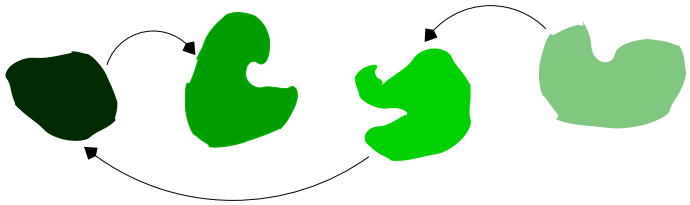


Figure 13.3: Islands of the evolutionary algorithm. Each PE has its own population and performs combine and mutation operations using different random seeds. From time to time the locally best partition is send to other PEs. Source: [118].

PEs used. A communication step is organized in rounds. In each round, a PE chooses a communication partner and sends her the currently best partition P of the local population. The selection of the communication partner is done uniformly at random among the eligible PEs. For a PE, a communication partner p' is called *eligible* if P has not been sent to p' in a previous round. Afterwards, a PE checks if there are incoming individuals and inserts them into the local population using the eviction strategy described above. If P is improved, all PEs are again eligible. This is repeated $\log p$ times. The principle is visualized in Figure 13.3.

Note that the algorithm can be implemented *completely asynchronously*, i.e. there is no need for a global synchronization. The process of creating individuals is parallelized as follows: Each PE makes $s' = |S|/p$ calls to KaFFPa using different seeds to create s' individuals. Afterwards we do the following $S - s'$ times: the root PE computes a random cyclic permutation of all PEs and broadcasts it to all PEs. Each PE then sends a random individual to its successor in the cyclic permutation and receives an individual from its predecessor in the cyclic permutation. We call this particular part of the algorithm *quick start*.

Algorithm 15 LocallyEvolve

```
estimate population size S
while time left
  if elapsed time < t_total/f then
    create individual
    insert into local population
  else
    flip coin c with corresponding probabilities
    if c shows head then perform a mutation operation
    else perform a combine operation
    insert offspring into population if possible
  communicate according to communication protocol
```

13.3 Meta-Heuristics for Graph Clustering

Ensemble learning is a paradigm in machine learning, where several intermediate classifiers (called weak or base classifiers) are generated and combined to get a stronger single classifier. The algorithms used to compute the weak classifiers are called weak learners. An important notion is, that even if a weak learner has only a slightly better accuracy than random choice, by combining several classifiers created by this weak learner, a strong classifier can be created.

Two examples of ensemble learning strategies are bagging and boosting. A bagging algorithm for supervised classification trains several classifiers from bootstraps of the training data. The combined classifier is computed by simple majority voting of the ensemble of base classifiers, i.e. a data item gets the label the majority of base classifiers assigns to that data item. A simple boosting algorithm works with classifiers trained from three subsets of the training data. Another ensemble learning strategy is Stacked Generalization. This strategy is based on the assumption that some data points are more likely to be misclassified than others, because they are near to the boundary that separates different classes of data points. First, an ensemble of classifiers is trained. Then, using the output of the classifiers a second level of classifiers is trained with the outputs of the ensemble of classifiers. In other words, the second level of classifiers learns for which input a first level classifier is correct or how to combine the "guesses" of the first level classifiers. The section is based on [104].

13.3.1 Core Groups Graph Clustering Scheme

We now restrict our considerations to the problem of whether a pair of vertices should belong to the same cluster or to different clusters. Making this decision is complicated. Many algorithms get misled during their search so that sometimes bad decisions are made. But what if we have one or more algorithms that find several clusterings with fair quality but still a lot of non-optimal decisions on whether a pair of vertices belongs to the same cluster? If all clusterings agree on whether a pair of vertices belongs to the same cluster, we can be pretty sure that this decision is correct. The CGGC scheme is based on this considerations [104]. The authors use the agreements of several clusterings with fair quality to decide whether a pair of vertices should belong to the same cluster. The groups of vertices which are assigned to the same cluster in every clustering (i.e. the maximal overlaps of the clusterings) are denoted as core groups. To abstract from any specific quality measure, we use the term good partition for a partition that has a good quality according to an arbitrary quality measure. The CGGC scheme consists of the following steps:

1. Create a set S of k good clusterings of G with base algorithm $A_{initial}$
2. Identify the partition $\hat{P}$ of the maximal overlaps in S
3. Create a graph $\hat{G}$ induced by the partition $\hat{P}$
4. Use base algorithm A_{final} to search for a good partition of $\hat{G}$
5. Project partition of $\hat{G}$ back to G

Initially, a set S of k clusterings of G is created. That means, a non-deterministic clustering algorithm is started k times to create the graph clusterings, k deterministic but different algorithms are used or a combination of both is used. In terms of ensemble learning, the used algorithms are the base algorithms or weak learners and the computed clusterings are the weak classifiers. Next, we combine the information of the weak classifiers: We calculate the maximal overlap of the clusterings in S. Let $c_p(v)$ denote the cluster that vertex v belongs in partition P. We create from a Set S of partition P_1, ..., P_k, of V a new partition $\hat{P}$ of V so that

$$\forall i \in [1,k], v,w \in V : c_{P_i}(w) = c_{P_i}(v) \Rightarrow c_{\hat{P}_i}(v) = c_{\hat{P}_i}(w)$$
$$\forall i \in [1,k], v,w \in V : c_{P_i}(w) \neq c_{P_i}(v) \Rightarrow c_{\hat{P}_i}(v) \neq c_{\hat{P}_i}(w)$$

Extracting the maximum overlap of an ensemble of partitions creates an intermediate solution which is used as the starting point for the base algorithm A_{final} to calculate the final clustering. The base algorithm used in this phase could be an algorithm used in step 1 or any other algorithm appropriate to optimize the objective function. For example, algorithms that are able to cluster the original network in reasonable time could be used to cluster the smaller graph $\hat{G}$ = ($\hat{V}$,$\hat{E}$) induced by $\hat{P}$. To create the induced graph, all vertices in a cluster in $\hat{P}$ are merged to one vertex in $\hat{G}$. Accordingly, $\hat{G}$ has as many vertices as there are clusters in $\hat{P}$. An edge $(v,w) \in \hat{E}$ has the height of the combined weights of all edges in G that connect vertices in the clusters represented by v and w. Then, the clustering of $\hat{G}$ would have to be projected back to G to get a clustering of the original graph.

References

The chapter is based on Ovelgönne [104] and Schulz. It has been created by Sergey Hayrapetyan and Christian Schulz.

Chapter 14

Diffusion-based Graph Partitioning

The graph partitioning heuristics we have seen so far, aim to reduce the edge cut of the partitions. That is, the number of edges adjacent to nodes in different partitions. In this chapter, the objectives of the partitioning are taken from the load balancing research field, where the edge cut is not the best measure of a good partitioning. In the parallelization of numerical simulation algorithms, all processors should roughly contain the same amount of computational work, while making the communication between the processors as small as possible. Thus, the number of boundary vertices in the partitions play an important role, because they represent data dependencies, and the purpose of this partitioning is to minimize the partition boundaries.

This problem can be solved by existing partitioners, like multilevel algorithms, which create a contraction hierarchy, find an initial partitioning of the smallest graph, and refine the partitions at each step back to the initial graph. While these heuristics minimize the edge-cut, we describe an approach in this chapter which does not. Instead, we apply a diffusive process inside a learning framework for partitioning a graph. Diffusive processes are well studied for load balancing in Finite Element Simulation methods. Moreover, since other graph partitioning heuristics are hard to parallelize, the approach presented in this chapter shows natural parallelism.

14.1 The Bubble Framework

The previously presented partitioners aim to minimize edge cut. This is not necessarily the best metric. A more appropriate metric is the number of boundary vertices. It is used in processor load balancing problems, where a small number of boundary vertices produces a small communication volume between the processing units. The following describes a shape optimizing load balancer. Shape optimized

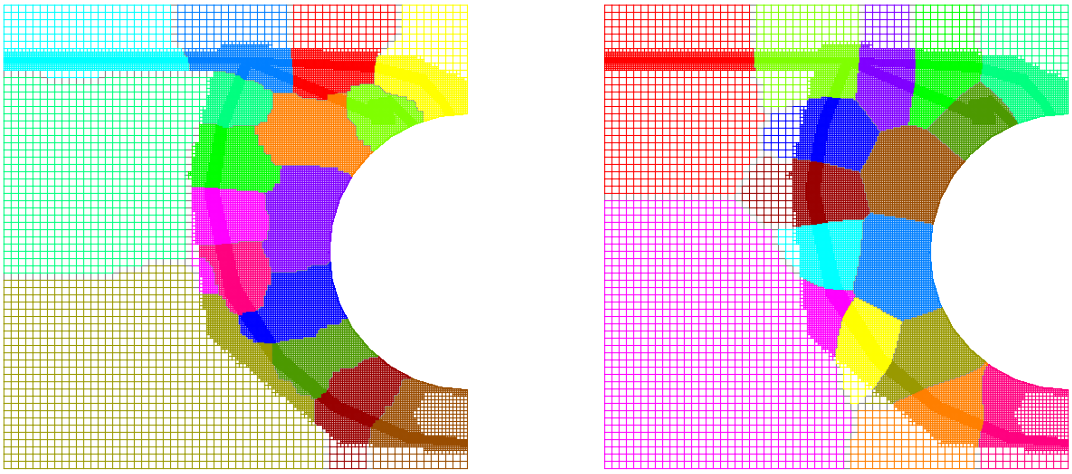


Figure 14.1: Partitioning the shock graph after 9 refinement steps into 16 partitions. Metis (left) computes a solution with edge-cut 1226 and 2082 boundary vertices, while the shape optimizing approach (right) finds a partitioning with edge-cut 1168 and only 1795 boundary vertices.

partitions typically exhibit few boundary vertices. Figure 14.1 gives a visualization of a partitioning computed with the edge-cut minimizing library Metis and a solution obtained with the new approach.

In this section we describe a learning framework which was inspired by the k-means algorithm in machine learning. It first starts with an initial, often randomly chosen vertex (seed) for each partition. It then adds adjacent vertices to each partition, thereby growing them, until their borders collide, continuing to grow them along this border "just like soap bubbles". After the whole graph is covered with the partitions, i.e. every vertex belongs to a partition, new centers for each partition are computed. Figure 14.2 illustrates this approach. This is usually repeated until a stable state, where the movement of all seeds is small enough, is reached.

The three steps can be implemented in several ways. In the first step, the seeds are chosen randomly. Then, the partitions are grown with a bread-first-like algorithm for every seed. In the last step, the new seed is chosen as being the vertex which minimizes the maximum distance to any other vertex in its partition.

This approach has several disadvantages. The initial position of the seeds might be bad, and more iterations are necessary to compute the final partitions. Also, the partition sizes vary extremely and the time to find the new seeds is long, because a breadth-first-search has to be solved for every vertex. The partition shape is also unsatisfactory. Another important disadvantage is that the growth phase cannot be parallelized because vertices are assigned serially and earlier assignments have a

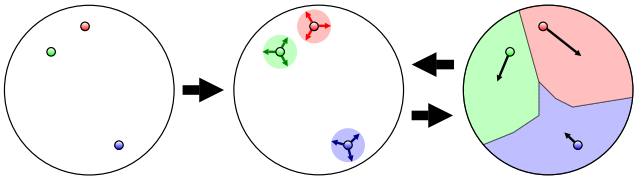


Figure 14.2: The three operations of the learning bubble growing scheme: Init: Determination of initial seeds for each partition (left). Grow: Growing around the seeds (middle). Move: Movement of the seeds to the partition centers (right).

great impact on later decisions.

In a second approach, the seeds are distributed more evenly over the graph. To grow the partitions, the smallest partition with at least one adjacent unassigned vertex grabs the vertex with the smallest Euclidean distance to its seed. The new seed of a partition is determined as the vertex for which the sum of Euclidean distances to all other vertices of the same partition is minimal. To find this vertex quickly, some estimation is used.

This approach solves some of the previous problems, in particular the seed distribution and the partition balance, because the smallest possible partition receives the next vertex. By including coordinates in the choice of the next vertex, the partitions are usually also geometrically well shaped (and connected), which is the main goal of this approach. Other quality metrics are not considered. However, by relying on vertex coordinates, this approach only works if the euclidean distance between somewhat coincides with the length between vertices. This approach also breaks when there are "holes" in the graph. The situation with parallelization stays the same, since the choice about the next vertex to include is still strictly sequential.

14.2 Diffusive Mechanisms

We solve the problems stated in the previous section by applying a diffusive mechanism to the steps of the Bubble framework. Diffusion is used in physics to model transport phenomena such as heat flows and also, as mentioned before, to model load distribution in parallel numerical computing. The main idea behind applying diffusion to graph partitioning is the fact that load primarily diffuses into densely connected regions of the graphs rather than into sparsely connected ones. As a consequence, one can expect to identify vertex sets that possess a high number of internal and a small number of external edges. Furthermore, diffusion possesses a large amount of parallelism since it performs local operations only.

The growth and seed determination processes are explained in the following. First, an initial load is placed on the seed vertex of every partition and a diffusive

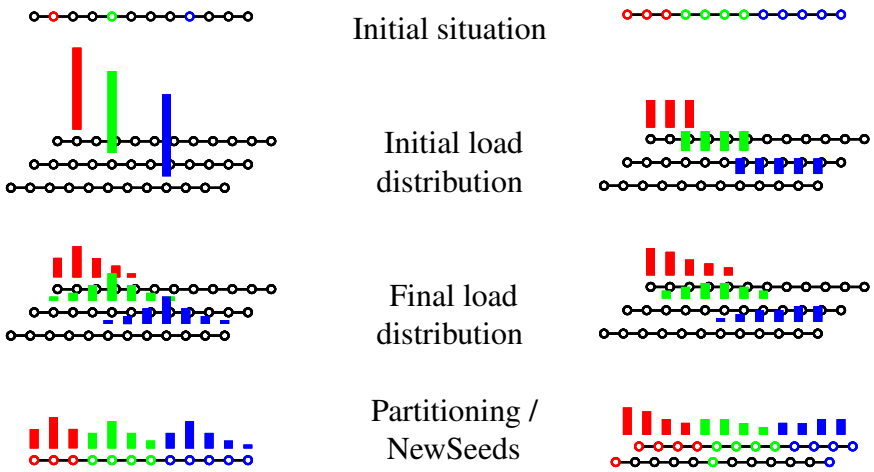


Figure 14.3: Schematic view: Placing load on single vertices (left) or a partition (right), the diffusion process and the mapping of the vertices to the partitions according to the load.

process is used to spread it into the graph. This is performed independently for every partition. After the load is distributed, we assign each vertex to that partition it has obtained the highest load amount from. This is illustrated in figure 14.3 (left). The next step (right) does not place load on a single vertex but distributes it evenly among all vertices of the given partitions. After performing the diffusion process, the resulting load distribution can either be used for an optional consolidation step or for contracting the partitions to the seed vertices of the next iteration. A consolidation again assigns the vertices to partitions according to the highest load as in the previous step. This further improves the partition shapes. During a contraction, for each partition the vertex containing the highest load becomes its new seed.

The diffusion process must possess two important properties:

1. **"Hill-like"**: To guarantee connected partitions, the load distribution function must be "hill-like". That means, that vertices closer to the partition center receive more load than those further away.

2. **Connectivity**: The load should spread inside well-connected parts of the graph, thereby moving the new partition centers towards denser regions, producing partitions with boundaries in sparser regions. This reduces the number of boundary vertices and therefore improves the partition quality. Thus, load must not only be distributed according to the distance in the graph, but according to connectivity.

This approach was implemented in the First Order Diffusion Scheme (FOS),

which converges towards a fully balanced load situation. It must be interrupted at some point to preserve the hill-like structure. Though it is possible to determine a suitable number of iterations, it is rather difficult and since the interruption point depends both on the graph and on the number of partitions, the authors were not able to find a general rule for this, which results in an unreliable implementation.

To avoid this, the disturbed diffusion scheme FOS/A was developed, which – based on FOS – shifts load back in each iteration from all non-empty vertices to the seed, preserving the hill-like structure in the final state. According to all experiments, this diffusion scheme converges, although no proof has been found yet. Due to the disturbance, it performs slower than FOS.

Another approach computes a hill-like load distribution by adding an extra vertex to the graph, which is connected to every other vertex, and shifts load according to the new edges' capacity ϕ. Varying this parameter controls the spreading of the load. The load distribution is computed by solving a system of linear equations.

The technique presented in the following omits the extra vertex and shows that running the algorithm with $\phi = 0$ is indeed equivalent to applying a disturbed diffusion scheme called FOS/C.

14.2.1 Prerequisites

To show some properties of the new diffusion scheme, we first need to introduce some basic notations. The incidence matrix A of a graph $G = (V, E)$ is $A \in \{-1, 0, +1\}^{|V|\times|E|}$. A contains in each column corresponding to edge $e = (u, v)$ the entries -1 and $+1$ in the rows u and v, and 0 elsewhere. The Laplacian matrix $L \in \mathbb{Z}^{|V|\times|V|}$ is defined as $L = AA^T$. In other words, $L_{u,u} = \deg(u)$, $L_{u,v} = -1$ for $\{u, v\} \in E$ and $L_{u,v} = 0$ otherwise. The Laplacian matrix is symmetric, positive semi-definite, and of rank $|V| - 1$. The matrix $M = I - \alpha L$ (with $0 < \alpha < 1$ suitably chosen, e.g. $\alpha = 1/(\text{maxdeg}(G) + 1)$) is a *diffusion matrix*, since it is non-negative, symmetric and doubly stochastic. Then, $1 = \mu_1 > \mu_2 \geq \cdots \geq \mu_n > -1$ are the eigenvalues of M and $\mathbf{1}$ is the eigenvector corresponding to μ_1. By $\mathbf{1}$ we denote the vector whose components are all 1.

Furthermore, we write the workloads at time t as an n-vector $w^{(t)}$ where $w_i^{(t)}$ is the number of tasks to be done by processor i at time t. Thus, in order to achieve a balanced workload for all processing units, a node v must receive roughly $\alpha(w_u^{(t)} - w_v^{(t)})$ of the workload from every other adjacent processor $u \neq v$, with α as above. If this difference is positive, then node v receives precisely this amount of work from u. If it is negative, the workload transfer is in the opposite direction. We can now formally introduce the First Order Diffusion Scheme.

Definition 7 (FOS)
Given a connected graph $G = (V, E)$ and a suitable constant α. In each iteration, the first order scheme (FOS) performs the operations:

$$\begin{aligned} f^{(i)}_{e=(u,v)} &= \alpha \cdot \left(w^{(i)}_u - w^{(i)}_v\right) \\ w^{(i+1)}_v &= w^{(i)}_v - \sum_{e=(v,*)} f^{(i)}_e \end{aligned} \tag{14.1}$$

As described earlier, we can think about the quantity $f^{(i)}_{e=(u,v)}$ as the amount of work that is transferred from node u to node v at a moment i in time, in order to achieve a balanced workload throughout the network. The constant α is chosen as above. Equation 14.1 then updates the workload of a node v according to the workload of all its neighbours. In matrix notation, FOS can be written as

$$w^{(i+1)} = Mw^{(i)}$$

where $M = I - \alpha L$ is the diffusion matrix.

As already mentioned, FOS converges towards the equally balanced load situation $\bar{w} = \sum_i w_i/n$. Furthermore, we know that the computed flow is minimal according to the $\|\cdot\|_2$-norm.

Lemma 35 (FOS convergence)
Let $M = I - \alpha L$ be the FOS diffusion matrix, $w^{(0)}$ the initial and $\bar{w}$ the balanced load situation. Then, the first order scheme $w^{(i+1)} = Mw^{(i)}$ converges to $\bar{w}$.

L does not have full rank. Hence, the existence of a solution depends on the right hand side of the linear equation.

Lemma 36 (Existence of a solution)
The equation $Lw = d$ has a solution (and then infinitely many), iff $d \perp \mathbf{1}$.

The next lemma states that the $\|\cdot\|_2$-minimal flow can be computed by solving a linear system.

Lemma 37
Consider the quadratic minimization problem

$$\min! \, \|f\|_2 \text{ with respect to } Af = d$$

Provided that $d \perp 1$, the solution to this problem is given by

$$f = A^T w, \text{ where } Lw = d$$

To prove the convergence of FOS/C, we require the following observation.

Lemma 38
*Let M be a diffusion matrix and d be a vector perpendicular to **1**. Then,*

$$\lim_{i\to\infty}(I + M + M^2 + \cdots + M^i)d = (I - M)^{-1}d$$

Proof. Recall that **1** is an eigenvector to the simple eigenvalue 1 of M. Since $d \perp 1$, i.e. $\sum_{j=1}^{n} d_j = 0$, it follows that $\lim_{i\to\infty} M^{i+1}d = 0$. Hence,

$$\begin{aligned}&\lim_{i\to\infty}(I - M)(I + M + M^2 + \cdots + M^i)d\\ =\ &\lim_{i\to\infty}(I - M^{i+1})d = \lim_{i\to\infty} d - M^{i+1}d\\ =\ &d\end{aligned}$$

Therefore, $(I + M + M^2 + \cdots + M^i)$ is the inverse to $(I - M)$ for $i \to \infty$ and any vector d perpendicular to **1**, so that the claim follows. □

14.2.2 Diffusion with Constant Draining

We are now introducing a new diffusion scheme. This scheme is based on FOS, but is disturbed in every iteration. In contrast to FOS/A, this disturbance is not restricted to the non-empty vertices, but performed on all vertices.

In contrast to genuine FOS, the FOS/C scheme performs two operations in each iteration. While the first one is the original diffusion step, the second step introduces a disturbance by shifting a small load amount $\delta > 0$ from all vertices of the graph to some selected source vertices $S \subset V$. This disturbance can be described by the drain vector $d \in \mathbb{R}^n$, which is defined as

$$d_v = \begin{cases} -\delta & : v \notin S \\ \delta \cdot |V|/|S| - \delta & : \text{otherwise} \end{cases}$$

The vector d is added to the load vector resulting from the diffusion step. Note that, since $\langle d, \mathbf{1}\rangle = 0$, this does not change the total amount of system load.

Definition 8 (FOS/C)
Given a connected graph $G = (V, E)$ and a suitable constant α. Let $\delta > 0$ be the drain constant and d the corresponding drain vector. Let $S \subset V$ be the set of source vertices. In iteration i, $w_v^{(i)}$ denotes the load on vertex v and $f_e^{(i)}$ the flow over edge e. Let $w^{(0)}$ represent the initial load situation. In each iteration i, the FOS/C scheme performs the computations:

$$\begin{aligned} f_{e=(u,v)}^{(i)} &= \alpha \cdot \left(w_u^{(i)} - w_v^{(i)}\right)\\ w_v^{(i+1)} &= w_v^{(i)} - \sum_{e=(v,*)} f_e^{(i)} + d_v \end{aligned}$$

In matrix notation, FOS/C can be written as

$$w^{(i+1)} = Mw^{(i)} + d.$$

Theorem 39 (Convergence of FOS/C)
The FOS/C scheme converges for any arbitrary initial load vector $w^{(0)}$.

Proof. Repeatedly applying the diffusion matrix to the initial load vector $w^{(0)}$, we obtain

$$\begin{aligned} w^{(1)} &= Mw^{(0)} + d \\ w^{(2)} &= Mw^{(1)} + d = M(Mw^{(0)} + d) + d \\ &= M^2w^{(0)} + (M + I)d \\ &\vdots \\ w^{(i)} &= M^iw^{(0)} + (M^{i-1} + \cdots + M + I)d \end{aligned}$$

Due to lemma 38, this yields

$$\begin{aligned} w^{(\infty)} &= M^\infty w^{(0)} + (I - M)^{-1}d \\ &= M^\infty w^{(0)} + (\alpha L)^{-1}d \end{aligned}$$

□

Since $M^\infty w^{(0)}$ is the evenly balanced load that FOS computes, one can see that the solution of the disturbed scheme FOS/C could also be determined by solving a system of linear equations. In fact, in the converged state, all load that is moved back onto the source vertices has to be sent back in one iteration step. This means that the solution of FOS/C is equivalent to computing a $\|\cdot\|_2$-minimal flow from the source vertices into the graph.

Corollary 3
The convergence state $w^{(*)}$ of FOS/C can be characterized as:

$$\begin{aligned} w^{(*)} &= Mw^{(*)} + d \\ \Leftrightarrow (I - M)w^{(*)} &= d \\ \Leftrightarrow \alpha Lw^{(*)} &= d \end{aligned}$$

Hence, the convergence state can be determined by solving the system of linear equations $Lw = d$, where $w = \alpha w^{(*)}$.

Since the solution of $Lw = d$ is only determined up to a constant, we choose the one with $\sum_{v\in V} w_v^{(*)} = 0$. This normalization also ensures that the load distributions computed for each partition have a common reference point and are therefore comparable.

Definition 9
If $|S| = 1 (|S| > 1)$, we call FOS/C iteration to the steady state or its corresponding linear system a single-source (multiple-source) FOS/C procedure. *Also, let $[w^{(t)}]_v^u$ ($[w]_v^u$) denote the load on node v in time step t (in the steady state) of a single-source FOS/C procedure with node u as source.*

Definition 10
A random walk *on a graph $G = (V, E)$ is a discrete time stochastic process, which starts on an initial node and performs the following step in each iteration. It chooses one of the neighbors of the current node randomly and then proceeds to the neighbor just chosen to start the next iteration. The transition probabilities are given by a stochastic transition matrix P, whose entry (u, v) denotes the probability to move from node u to node v. The random walk may stay on the current node v with positive probability if $P_{vv} > 0$.*

Remark: $[w]_v^u = \lim_{t\to\infty} ([M^t w^{(0)}]_v^u + n\delta(\sum_{l=0}^{t-1} M_{v,u}^l) - t\delta)$, where $M_{v,u}^l$ is the probability of a random walk (defined by the stochastic diffusion matrix M) starting at v to be on u after l steps. Since $[M^t w^{(0)}]_v^u$ converges towards the balanced load distribution, the important part of a FOS/C load in the steady state is $\sum_{l=0}^{\infty} M_{v,u}^l$, which is the sum of transition probabilities of random walks with increasing lengths.

14.2.3 Bubble-FOS/C with Algebraic Multigrid

Bubble-FOS/C implements the operations of the Bubble framework with FOS/C procedures, single-source ones for `AssignPartition` and multiple-source ones for `ComputeCenters`. Its outline is shown in Algorithm 16, where $\Pi = \{\pi_1, \ldots, \pi_k\}$ denotes the set of partitions and $Z = \{z_1, \ldots, z_k\}$ the set of the corresponding center nodes. First, the algorithm determines pairwise disjoint initial centers (line 1), which can be done in an arbitrary manner. After that, with the new centers at hand, the main loop is executed. It determines in alternating calls a new partitioning (`AssignPartition`, lines 4 – 10) and new centers (`ComputeCenters`, lines 12 – 15). The loop can be iterated until convergence is reached or, if running time is important, a constant number of times.

It turns out that this iteration of two alternating operations yields very good partitions. The ability of distinguishing sparsely from densely connected components can be explained by FOS/C's connection to random walks pointed out above. As random walks tend to stay a long time within a dense region once they have reached it, Bubble-FOS/C usually obtains partition centers in dense regions and boundaries tend to be in sparse ones (as desired). Moreover, since the isolines of the FOS/C load in the steady state tend to have a circular shape, the final partitions are very compact and have short boundaries.

Most work performed by Bubble-FOS/C consists in solving linear systems. It is therefore necessary to employ a very efficient solver. Among such solvers are Multigrid methods and the Algebraic multigrid (AMG) extension case. The latter constructs a multilevel hierarchy based on weighted interpolation with a carefully chosen set of nodes for the coarser level. AMG is used here as a linear solver since the same system matrix L is used repeatedly, so that the hierarchy construction is amortized. Furthermore, as an AMG hierarchy is also a sequence of coarser graphs retaining the structure of the original one, we use it in our Bubble-FOS/C implementation for providing a multilevel hierarchy (instead of the standard matching approach). For Bubble-FOS/C this alternative hierarchy construction method hardly influences the solution quality, but speeds up computations significantly.

Algorithm 16 Sketch of the main Bubble-FOS/C algorithm

Require: Graph G, partition number k
Ensure: Resulting partition Π
1: $Z \leftarrow \text{InitialCenters}(G, k)$ /* Arbitrary disjoint initial centers */
2: **for** $\tau = 1, 2, \ldots$ **until** convergence **do**
3: /* `AssignPartition` */
4: **parallel for** each partition π_c **do**
5: Initialize $d_c(S = \{z_c\})$, solve and normalize $Lw_c = d_c$
6: /* after synchronization: update Π */
7: **end for**
8: **for** each node $v \in \pi_c$ **do**
9: $\Pi(v) = p : w_p(v) \geq w_q(v) \forall q \in \{1, \ldots, k\}$
10: **end for**
11: /* `ComputeCenters` */
12: **parallel for** each partition π_c **do**
13: Initialize $d_c(S = \{z_c\})$ and solve $Lw_c = d_c$
14: $z_c = \arg\max_{v \in \pi_c} w_c(v)$
15: **end for**
16: **end for**
17: **return** Π

14.3 A Faster Approach

The Bubble-FOS/C scheme takes a relatively long computation time due to the fact that the linear equation system is solved for the whole graph on every iteration. An improvement in the running time can be achieved by considering load exchange

only at the partition boundaries. In [93] the authors examine this fact and build a multilevel hierarchy, applying this observation.

References

Most of this chapter is based on [94] and [93]. Some more detailed insights about the diffusion scheme were taken from [36]. This chapter was written by Robert Hangu.

Bibliography

[1] M. E. J. Newman A. Clauset and C. Moore. Finding community structure in very large networks. *The American Physical Society*, 2004.

[2] J. Abello, A. L. Buchsbaum, and J. Westbrook. A Functional Approach to External Graph Algorithms. *Algorithmica*, 32(3):437–458, 2002.

[3] Y. Akhremtsev, P. Sanders, and C. Schulz. (Semi-)External Algorithms for Graph Partitioning and Clustering, booktitle = Proceedings of the 17th Workshop on Algorithm Engineering and Experiments, ALENEX 2015. pages 33–43, 2015.

[4] R. Andersen and K. J. Lang. An Algorithm for Improving Graph Partitions. In *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms*, pages 651–660. Society for Industrial and Applied Mathematics, 2008.

[5] K. Andreev and H. Räcke. Balanced Graph Partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[6] L. Arge. The Buffer Tree: A Technique for Designing Batched External Data Structures. *Algorithmica*, 37(1):1–24, 2003.

[7] V. Arnau, S. Mars, and I. Marín. Iterative cluster analysis of protein interaction data. *Bioinformatics*, 21(3):364–378, 2005.

[8] G. Ausiello. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 1999.

[9] D. Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13(4):475–493, 1983.

[10] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. PhD thesis, 1996.

[11] S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. In *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pages 711–718, 1993.

[12] U. Benlic and J. K. Hao. An Effective Multilevel Memetic Algorithm for Balanced Graph Partitioning. In *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence*, pages 121–128, 2010.

[13] U. Benlic and J. K. Hao. A Multilevel Memetic Approach for Improving Graph k-Partitions. *IEEE Transactions on Evolutionary Computation*, 15(5):624–642, 2011.

[14] U. Benlic and J. K. Hao. An Effective Multilevel Tabu Search Approach for Balanced Graph Partitioning. *Computers & Operations Research*, 38(7):1066–1075, 2011.

[15] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[16] M. J. Berger and S. H. Bokhari. A partitioning strategy for pdes across multiprocessors. In *ICPP*, pages 166–170, 1985.

[17] M. J. Berger and S. H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Transactions on Computers*, 100(5):570–580, 1987.

[18] C. E. Bichot. A New Method, the Fusion Fission, for the Relaxed k-Way Graph Partitioning Problem, and Comparisons with some Multilevel Algorithms. *Journal of Mathematical Modelling and Algorithms*, 6(3):319–344, 2007.

[19] C.E. Bichot and P. Siarry. *Graph Partitioning*. Iste Series. Wiley, 2011.

[20] V. D. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008.

[21] K. D. Boese, A. B. Kahng, and S. Muddu. A New Adaptive Multi-Start Technique for Combinatorial Global Optimizations. *Operations Research Letters*, 16(2):101–113, 1994.

[22] P. Bonsma. Most Balanced Minimum Cuts. *Discrete Applied Mathematics*, 158(4):261–276, 2010.

[23] R. B. Boppana. Eigenvalues and Graph Bisection: An Average-Case Analysis (Extended Abstract). In *Proceedings of the 28th Symposium on Foundations of Computer Science*, pages 280–285, 1987.

[24] U. Brandes, M. Gaertler, and D. Wagner. Engineering graph clustering: Models and experimental evaluation. *ACM Journal of Experimental Algorithmics*, 12, 2007.

[25] A. Brandt. Multiscale Scientific Computation: Review 2001. In *Multiscale and Multiresolution Methods*, volume 20 of *LNCSE*, pages 3–95. Springer, 2002.

[26] W. L. Briggs and S. F. McCormick. *A Multigrid Tutorial*, volume 72. Society for Industrial Mathematics, 2000.

[27] R. Brillout. *A Multi-Level Framework for Bisection Heuristics*. PhD thesis, Karlsruhe, Baden Wiirttemberg, Germany Karlsruhe Institute of Technology, 2009.

[28] T. Bui, S. Chaudhuri, F. Leighton, and M. Sipser. Graph Bisection Algorithms with Good Average Case Behavior. *Combinatorica*, 7:171–191, 1987.

[29] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering – Selected Topics,* to app.*, ArXiv:1311.3144*, 2014.

[30] Ü. V. Çatalyürek and C. Aykanat. Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication. In *Proceedings of the 3rd International Workshop on Parallel Algorithms for Irregularly Structured Problems*, volume 1117, pages 75–86. Springer, 1996.

[31] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed Systems*, 10(7):673–693, Jul 1999.

[32] P. Chardaire, M. Barake, and G. P. McKeown. A PROBE-Based Heuristic for Graph Partitioning. *IEEE Transactions on Computers*, 56(12):1707–1720, 2007.

[33] J. Chen and I. Safro. Algebraic Distance on Graphs. *SIAM Journal on Scientific Computing*, 33(6):3468–3490, 2011.

[34] B. V. Cherkassky and A. V. Goldberg. Negative-Cycle Detection Algorithms. In *Proceedings of the 4th European Symposium on Algorithms*, volume 1136 of *LNCS*, pages 349–363, 1996.

[35] C. Chevalier and I. Safro. Comparison of Coarsening Schemes for Multilevel Graph Partitioning. In *Proceedings of the 3rd International Conference on Learning and Intelligent Optimization*, volume 5851 of *LNCS*, pages 191–205, 2009.

[36] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of parallel and distributed computing*, 7(2):279–301, 1989.

[37] K. A. De Jong. *Evolutionary Computation: A Unified Approach*. MIT Press, 2006.

[38] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms*, volume 6630 of *LCNS*, pages 376–387. Springer, 2011.

[39] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *Proceedings of the 25th International Parallel and Distributed Processing Symposium*, pages 1135–1146, 2011.

[40] R. Diekmann, B. Monien, and R. Preis. Using Helpful Sets to Improve Graph Bisections. *Interconnection Networks and Mapping and Scheduling Parallel Computations*, 21:57–73, 1995.

[41] B. Doerr and M. Fouz. Asymptotically Optimal Randomized Rumor Spreading. In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming, Proceedings, Part II*, volume 6756 of *LNCS*, pages 502–513. Springer, 2011.

[42] W. E. Donath and A. J. Hoffman. Algorithms for Partitioning of Graphs and Computer Logic Based on Eigenvectors of Connection Matrices. *IBM Technical Disclosure Bulletin*, 15(3):938–944, 1972.

[43] W. E. Donath and A. J. Hoffman. Lower Bounds for the Partitioning of Graphs. *IBM Journal of Research and Development*, 17(5):420–425, 1973.

[44] D. Drake and S. Hougardy. A Simple Approximation Algorithm for the Weighted Matching Problem. *Information Processing Letters*, 85:211–213, 2003.

[45] S. Dutt. New Faster Kernighan-Lin-type Graph-Partitioning Algorithms. In *Proceedings of the 4th IEEE/ACM International Conference on Computer-Aided Design*, pages 370–377, 1993.

[46] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Conference on Design Automation*, pages 175–181, 1982.

[47] M. Fiedler. A Property of Eigenvectors of Nonnegative Symmetric Matrices and its Application to Graph Theory. *Czechoslovak Mathematical Journal*, 25(4):619–633, 1975.

[48] J. Fietz, M. Krause, C. Schulz, P. Sanders, and V. Heuveline. Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries. In *Proceedings of Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 818–829. Springer, 2012.

[49] L. R. Ford and D. R. Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[50] R. E. Tarjan G. D. Flake and K. Tsioutsiouliklis. Graph Clustering and Minimum Cut Trees. *Internet Mathematics*, 2004.

[51] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 434–443, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[52] P. Galinier, Z. Boujbel, and M. C. Fernandes. An Efficient Memetic Algorithm for the Graph Partitioning Problem. *Annals of Operations Research*, 191(1):1–22, 2011.

[53] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-Complete Problems. In *Proceedings of the 6th ACM Symposium on Theory of Computing*, STOC '74, pages 47–63. ACM, 1974.

[54] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.

[55] F. Glover. Tabu Search — Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[56] F. Glover. Tabu Search — Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[57] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[58] Gurobi Company. GUROBI. `http://www.gurobi.com/`. Accessed May 29, 2015.

[59] R. Görke. *An Algorithmic Walk from Static to Dynamic Graph Clustering*. PhD thesis, Karlsruhe Institute of Technology, 2010.

[60] B. Hendrickson. Graph Partitioning and Parallel Solvers: Has the Emperor No Clothes? In *Proceedings of the 5th International Symposium on Solving Irregularly Structured Problems in Parallel*, volume 1457 of *LNCS*, pages 218–225. Springer, 1998.

[61] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *Proceedings of the ACM/IEEE Conference on Supercomputing'95*. ACM, 1995.

[62] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.

[63] B. Hendrickson and E. Rothberg. Improving the Run Time and Quality of Nested Dissection Ordering. *SIAM J. on Scientific Computing*, 20(2):468–489, 1998.

[64] V. Henne, H. Meyerhenke, P. Sanders, S. Schlag, and C. Schulz. n-level hypergraph partitioning. *arXiv preprint arXiv:1505.00693*, 2015.

[65] J.-H. Hoepman. Simple distributed weighted matchings. *CoRR*, cs.DC/0410047, 2004.

[66] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a Scalable High Quality Graph Partitioner. *Proceedings of the 24th International Parallal and Distributed Processing Symposium*, pages 1–12, 2010.

[67] M. Holzer, F. Schulz, and D. Wagner. Engineering multi-level overlay graphs for shortest-path queries. In *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments, ALENEX 2006, Miami, Florida, USA, January 21, 2006*, pages 156–170, 2006.

[68] M. Holzrichter and S. Oliveira. A Graph Based Method for Generating the Fiedler Vector of Irregular Problems. In *Proceedings of the 11th Parallel and Distributed Processing Workshop*, volume 1586 of *LNCS*, pages 978–985. Springer, 1999.

[69] J. Hromkovič and B. Monien. The Bisection Problem for Graphs of Degree 4 (Configuring Transputer Systems). In *Proceedings of the 16th International Symposium on Mathematical Foundations of Computer Science*, pages 211–220. Springer, 1991.

[70] IBM. CPLEX Optimizer. `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html`. Accessed May 25, 2015.

[71] H. Inayoshi and B. Manderick. The Weighted Graph Bi-Partitioning Problem: A Look at GA Performance. *Parallel Problem Solving from Nature — PPSN III*, pages 617–625, 1994.

[72] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation. Part I, Graph Partitioning. *Operations Research*, 37(6):865–892, 1989.

[73] S. Jung and S. Pramanik. An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. *IEEE Trans. Knowl. Data Eng.*, 14(5):1029–1046, 2002.

[74] R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *J. ACM*, 51(3):497–515, 2004.

[75] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Trans. on Very Large Scale Integration VLSI Systems*, 7(1):69–79, 1999.

[76] G. Karypis and V. Kumar. Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs. In *Proceedings of the ACM/IEEE Conference on Supercomputing'96*, 1996.

[77] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[78] G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal on Parallel and Distributed Compututing*, 48(1):96–129, 1998.

[79] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(1):291–307, 1970.

[80] B.W. Kernighan. *Some Graph Partitioning Problems Related to Program Segmentation.* PhD thesis, Princeton, 1969.

[81] L. Khachiyan. A polynomial algorithm in linear programming. *Doklady Academiia Nauk SSSR*, 244:1093–1096, 1979.

[82] T. Kieritz, D. Luxen, P. Sanders, and C. Vetter. Distributed Time-Dependent Contraction Hierarchies. In *Proceedings of the 9th International Symposium on Experimental Algorithms*, volume 6049 of *LNCS*, pages 83–93. Springer, 2010.

[83] J. Kim, I. Hwang, Y. H. Kim, and B. R. Moon. Genetic Approaches for Graph Partitioning: A Survey. In *Proceedings of the 13th Annual Genetic and Evolutionary Computation Conference (GECCO'11)*, pages 473–480. ACM, 2011.

[84] J. M. Kleinberg. An Impossibility Theorem for Clustering. In *Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems, NIPS 2002, December 9-14, 2002, Vancouver, British Columbia, Canada]*, pages 446–453, 2002.

[85] B. Krishnamurthy. An Improved Min-Cut Algonthm for Partitioning VLSI Networks. *IEEE Transactions on Computers*, C-33(5):438–446, 1984.

[86] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, volume 8, pages 31–46, 2012.

[87] C. Lanczos. An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators. *Journal of Research of the National Bureau of Standards*, 45(4):255–282, 1950.

[88] K. Lang and S. Rao. A Flow-Based Method for Improving the Expansion or Conductance of Graph Cuts. In *Proceedings of 10th International Integer Programming and Combinatorial Optimization Conference*, volume 3064 of *LNCS*, pages 383–400. Springer, 2004.

[89] U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Proceedings of the Münster GI-Days*, 2004.

[90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., 1990.

[91] D. Luxen and D. Schieferdecker. Candidate Sets for Alternative Routes in Road Networks. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *LNCS*, pages 260–270. Springer, 2012.

[92] J. Maue and P. Sanders. Engineering Algorithms for Approximate Weighted Matching. In *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *LNCS*, pages 242–255. Springer, 2007.

[93] H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. *Journal of Parallel and Distributed Computing*, 69(9):750–761, 2009.

[94] H. Meyerhenke, B. Monien, and S. Schamberger. Accelerating Shape Optimizing Load Balancing for Parallel FEM Simulations by Algebraic Multigrid. In *Proceedings of 20th International Parallel and Distributed Processing Symposium*, 2006.

[95] H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning Complex Networks via Size-Constrained Clustering. In *Experimental Algorithms*, volume 8504 of *LNCS*, pages 351–363. Springer, 2014.

[96] H. Meyerhenke, P. Sanders, and C. Schulz. Parallel Graph Partitioning for Complex Networks. *Proceedings of the 29th International Parallal and Distributed Processing Symposium*, 2015.

[97] B. L Miller and D. E Goldberg. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Evolutionary Computation*, 4(2):113–131, 1996.

[98] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *Journal of Experimental Algorithmics (JEA)*, 11(2006), 2007.

[99] B. Monien and S. Schamberger. Graph Partitioning with the Party Library: Helpful-Sets in Practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, pages 198–205, 2004.

[100] M. EJ Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[101] M. EJ Newman and M. Girvan. Finding and Evaluating Community Structure in Networks. *Physical review E*, 69(2):026113, 2004.

[102] V. Osipov and P. Sanders. n-Level Graph Partitioning. In *Proceedings of the 18th European Conference on Algorithms: Part I*, volume 6346 of *LNCS*, pages 278–289. Springer, 2010.

[103] V. Osipov, P. Sanders, and C. Schulz. Engineering Graph Partitioning Algorithms. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*, volume 7276, pages 18–26. Springer, 2012.

[104] M. Ovelgönne. Distributed Community Detection in Web-Scale Networks. Technical report, 2012.

[105] D. A. Papa and I. L. Markov. chapter Hypergraph Partitioning and Clustering, pages 61–1–61–19. Chapman and Hall/CRC, 2007.

[106] B. N. Parlett. The Rayleigh Quotient Iteration and Some Generalizations for Nonnormal Matrices. *Mathematics of Computation*, 28(127):679–693, 1974.

[107] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, 1980.

[108] J. C. Picard and M. Queyranne. On the Structure of All Minimum Cuts in a Network and Applications. *Mathematical Programming Studies*, 13:8–16, 1980.

[109] A. Pothen, H. D. Simon, and K. P. Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(3):430–452, 1990.

[110] R. Preis. Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs. In *Proceedings of the 16th Symposium on Theoretical Aspects of Computer Science (STACS'99)*, volume 1563 of *LNCS*, pages 259–269. Springer, 1999.

[111] U. N. Raghavan, R. Albert, and S. Kumara. Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. *Physical Review E*, 76(3), 2007.

[112] E. Rolland, H. Pirkul, and F. Glover. Tabu Search for Graph Partitioning. *Annals of Operations Research*, 63(2):209–232, 1996.

[113] D. Ron, I. Safro, and A. Brandt. Relaxation-Based Coarsening and Multiscale Graph Organization. *Multiscale Modeling & Simulation*, 9(1):407–423, 2011.

[114] I. Safro, P. Sanders, and C. Schulz. Advanced Coarsening Schemes for Graph Partitioning. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *LNCS*, pages 369–380. Springer, 2012.

[115] L. A. Sanchis. Multiple-Way Network Partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989.

[116] P. Sanders. Fast Priority Queues for Cached Memory. *ACM Journal of Experimental Algorithmics*, 5:312–327, 1999.

[117] P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Proceedings of the 19th European Symposium on Algorithms*, volume 6942 of *LNCS*, pages 469–480. Springer, 2011.

[118] P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX'12)*, pages 16–29, 2012.

[119] P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, LNCS. Springer, 2013.

[120] K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. In *The Sourcebook of Parallel Computing*, pages 491–541, 2003.

[121] C. Schulz. Scalable Parallel Refinement of Graph Partitions. Diploma Thesis, Universität Karlsruhe, 2009.

[122] C. Schulz. *High Quality Graph Partititioning*. PhD thesis, Karlsruhe Institute of Technology, 2013.

[123] F. Schulz, D. Wagner, and K. Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics*, 5:12, 2000.

[124] R. Shamir, R. Sharan, and D. Tsur. Cluster Graph Modification Problems. In *Graph-Theoretic Concepts in Computer Science*, pages 379–390. Springer, 2002.

[125] J. Shi and J. Malik. Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):888–905, 2000.

[126] J. Šíma and S. E. Schaeffer. On the np-completeness of some graph cluster measures. In *SOFSEM 2006: Theory and Practice of Computer Science*, pages 530–537. Springer, 2006.

[127] H. D. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Computing Systems in Engineering*, 2(2):135–148, 1991.

[128] H. D. Simon and S. H. Teng. How Good is Recursive Bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, 1997.

[129] A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004.

[130] R. V. Southwell. Stress-Calculation in Frameworks by the Method of "Systematic Relaxation of Constraints". *Proceedings of the Royal Society of London. Series A-Mathematical and Physical Sciences*, 151(872):56–95, 1935.

[131] C. L. Staudt and H. Meyerhenke. Engineering High-Performance Community Detection Heuristics for Massive Graphs. In *Proceedings 42nd Conference on Parallel Processing (ICPP'13)*, 2013.

[132] U. Trottenberg and A. Schuller. *Multigrid*. Academic Press, Inc., 2001.

[133] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 507–516, New York, NY, USA, 2013. ACM.

[134] C. Walshaw. Multilevel Refinement for Combinatorial Optimisation Problems. *Annals of Operations Research*, 131(1):325–372, 2004.

[135] C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

[136] M. Yannakakis. Node-and edge-deletion np-complete problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 253–264. ACM, 1978.

[137] N. Zeh. I/O-efficient Graph Algorithms. *EEF Summer School on Massive Data Sets*, 2002.