Technische Universität Chemnitz

Ulrich Elsner

Graph Partitioning
A survey

Sonderforschungsbereich 393

Numerische Simulation
auf massiv parallelen Rechnern

Preprint SFB393/97-27, Dec. 1997

Contents

Notation

Below we list (most of) the symbols used in the text together with a brief explanation of their meaning.

$(\mathcal{V}, \mathcal{E})$		A (undirected) graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$.
$\mathcal{E}$	$=$	$\{e_{i,j} \mid$ there is an edge between v_i and $v_j\}$, the set of edges.
$\mathcal{V}$	$=$	$\{v_i \mid i = 1, \ldots, n\}$, the set of vertices.
$W_{\mathcal{E}}$	$=$	$\{w_e(e_{i,j}) \in \mathbb{N} \mid e_{i,j} \in \mathcal{E}\}$, weights of the edges in $\mathcal{E}$.
$W_{\mathcal{V}}$	$=$	$\{w_v(v_i) \in \mathbb{N} \mid v_i \in \mathcal{V}\}$, weights of the vertices in $\mathcal{V}$.
$\deg(v_i)$	$=$	$\big\lvert \{v_j \in \mathcal{V} \,\lvert\, e_{i,j} \in \mathcal{E}\} \big\rvert$ the degree of vertex v_i. (The number of adjacent vertices)
$L(\mathcal{G})$		The Laplacian matrix of the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

$$l_{ij} = \begin{cases} -1 & \text{if } i \neq j \text{ and } e_{i,j} \in \mathcal{E} \\ 0 & \text{if } i \neq j \text{ and } e_{i,j} \notin \mathcal{E} \\ d(v_i) & \text{if } i = j \end{cases}$$

$A(\mathcal{G})$		The adjacency matrix of the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

$$a_{ij} = \begin{cases} -1 & \text{if } i \neq j \text{ and } e_{i,j} \in \mathcal{E} \\ \;\;0 & \text{otherwise} \end{cases}$$

$\emptyset$		the empty set.
$\lvert\,.\,\rvert$		for $\mathcal{X}$ a set, $\lvert\mathcal{X}\rvert$ is the number of elements in $\mathcal{X}$.

$\|\,.\,\|_p$		for $x \in \mathbb{C}^n$ a vector, $\|x\|$ is the p-norm of x, $\|x\|_p = (\sum_{i=1}^n \|x_i\|^p)^{1/p}$.
$\|\,.\,\|$		$= \|\,.\,\|_2$, the Euklidian norm.
e	$=$	$(1, 1, \ldots 1)^T$.
$\mathcal{O}(f(n))$	$=$	$\{\, g(n) \mid$ there exists a $C > 0$ with $\|g(n)\| \leq C \cdot f(n)$ for all $n \geq n_0\}$, the Landau symbol.
$A \dot{\cup} B$	$=$	$A \cup B$ with $A \cap B = \emptyset$, disjoint union. The dot just emphasizes that the two sets are disjoint.
I_n, I		The identity matrix of size n. If the size is obvious from the context, I is used.
$\sigma(A)$		spectrum of A.
$\mathrm{diag}(w)$		a diagonal matrix with the components of the vector w on the diagonal.

1 The problem

1.1 Introduction

While the performance of classical (von Neumann) computers has grown tremendously, there is always demand for more than the current technology can deliver. At any given moment, it is only possible to put so many transistors on a chip and while that number increases every year, certain physical barriers loom on the horizon (simply put, an electron has to fit through each data path, the information has to travel a certain distance in each cycle) that cannot be be surpassed by the technology known today.

But while it is certainly important to pursue new methods and technologies that can push these barriers a bit further or even break them, what can be done to get bigger performance now?

Mankind has for eons used teamwork to achieve things that a single human being is incapable of (from hunting as a group to building the pyramids). The same idea can be used with computers: parallel computing. Invented or rather revived about 15 years ago, parallel computers achieve performances that are impossible or much to expensive to achieve with single processor machines. Therefore parallel computers of one architecture or another have become increasingly popular with the scientific and technical community.

But just as working together as a team does not always work efficiently (10 people cannot dig a post-hole 10 times as fast as one) and creates new problems that do not occur when one is working alone (how to coordinate the 10000 peasants working on the pyramid), obstacles unknown to single processor programmers occur when working on a parallel machine.

First, not all parts of a problem can be parallelized but some have to be executed serially. This limits the total possible speedup (Ahmdal's law [1]) for a given problem of constant size. Quite often though the sequential part of the work tends to be independent from the size of the input and so for a bigger problem (adequate for sharing the work among more processors) a higher speedup is possible. Problems of this kind are called scalable [43].

Second, the amount of work done has to be distributed evenly amongst the processors. This is even more necessary if during the course of the computation some data has to be exchanged or some process synchronized. In this case, some of the processors have to wait for others to finish. This *load-balancing* might be easy for some problems but quite complicated for others. The amount of work per processor might vary during the course of the computation, for example when some critical point needs to be examined closer. So perfect balance in one step does not have to mean balance in the next.

This could easily be avoided by moving some of the work to another, underutilized processor. But this always means communication between different processors. And communication is, alas, time-consuming.

This brings us to the third obstacle. In almost all "real-life" problems, the processors regularly need information from each other to continue their work. This has two effects. As mentioned above, these exchanges entail synchronization points, making it necessary for some of the processors to wait for others. And of course communication itself takes some time. Depending on the architecture, exchanging even one bit of information between two processors might be many times slower than accessing a bit in the processor itself. There might be a high startup time such that exchanging hundreds of bytes takes almost the same amount of time as exchanging one byte. It might or might not be possible that exchanges between two disjoint pairs of processors takes place at the same time.

While all this is highly machine-dependent, it is evident that it is desirable to partition the problem (or the data) in such a way that not only the work is distributed evenly but also that at the same time communication is minimized. As we will see, for certain problems this will lead to the problem of partitioning a graph. In its simplest form this means that we divide the vertices of a graph into equal sized parts such that the number of edges connecting these parts is minimal.

1.2 Graph Partitioning

First, some notations (cf. [34, 61]):

A graph $(\mathcal{V}, \mathcal{E})$ consists of a number of vertices $\mathcal{V} = \{v_i \mid i = 1, \ldots, n\}$, some of which are connected by edges in $\mathcal{E} = \{e_{i,j} = (i,j) \mid \text{ there is an edge between } v_i \text{ and } v_j\}$. Other names for vertices include nodes, grid points or mesh points.

In order to avoid excessive subscripts, we will sometimes equate i with v_i and (i,j) with $e_{i,j}$ if there is no danger of confusion .

Given a subset $\bar{\mathcal{V}} \subset \mathcal{V}$ of vertices, the *induced subgraph* $(\bar{\mathcal{V}}, \bar{\mathcal{E}})$ contains all those edges $\bar{\mathcal{E}} \subset \mathcal{E}$ that join two vertices in $\bar{\mathcal{V}}$.

Both vertices and edges can have weights associated with them.

$W_{\mathcal{E}} = \{w_e(e_{i,j}) \in \mathbb{N} \mid e_{i,j} \in \mathcal{E}\}$ are the weights of the edges and $W_{\mathcal{V}} = \{w_v(v_i) \in \mathbb{N} \mid v_i \in \mathcal{V}\}$ are the weights of the vertices.

The weights are non-negative integers. This often allows for a more efficient analysis or implementation of algorithms. For example, sorting a list of integers of restricted size can be done using the bucket-sort algorithm with its $\mathcal{O}(n)$ performance. But this restriction is not as big as it seems. For example, positive rational weights can easily be mapped to $\mathbb{N}_+$ by scaling them with the smallest common multiple of the denominators. If no weights are given, all weights should be considered to be 1.

The simplest form of graph partitioning, *unweighted graph bisection* is this: given a graph consisting of vertices and edges, divide the vertices in two sets of equal size such that the number of edges between these two parts is minimized. Or, more formally:

Let $G = (\mathcal{V}, \mathcal{E})$ be a graph with $|\mathcal{V}|$ (the number of of elements in $\mathcal{V}$) even.

Find a partition $(\mathcal{V}_1, \mathcal{V}_2)$ of $\mathcal{V}$ (i.e., $\mathcal{V}_1 \cup \mathcal{V}_2 = \mathcal{V}$ and $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$. This *disjoint* union ist sometimes expressed by $\mathcal{V}_1 \dot{\cup} \mathcal{V}_2 = \mathcal{V}$) with

$$|\mathcal{V}_1| = |\mathcal{V}_2|$$

such that

$$|\{e_{i,j} \in \mathcal{E} \mid v_i \in \mathcal{V}_1 \text{ and } v_j \in \mathcal{V}_2\}| \tag{1.1}$$

is minimized among all possible partitions of $\mathcal{V}$ with equal size.

A more general formulation involves weighted edges, weighted vertices and p disjoint subsets . Then the sum of weights of vertices in each of the subsets should be equal while the sum of the edges between these subsets is minimized. Again, more formally:

Let $G = (\mathcal{V}, \mathcal{E})$ be a graph with weights $W_{\mathcal{V}}$ and $W_{\mathcal{E}}$ (with $\sum_{v_i \in \mathcal{V}} w_v(v_i)$ divisible by p).

Find a p–partition $(\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_p)$ of $\mathcal{V}$, i.e.,

$$\bigcup_{i=1}^{p} \mathcal{V}_i = \mathcal{V} \quad \text{and} \quad \mathcal{V}_i \cap \mathcal{V}_j = \emptyset \quad \text{for all} \quad i \neq j$$

with

$$\sum_{v(i) \in \mathcal{V}_j} w_v(v_i) \quad \text{equal for all} \quad j \in \{1, 2, \ldots, p\}$$

such that

$$\sum_{\substack{e_{i,j} \in \mathcal{E} \text{ with} \\ v_i \in \mathcal{V}_p,\, v_j \in \mathcal{V}_q \text{ and } p \neq q}} w_e(e_{i,j}) \tag{1.2}$$

is minimized among all possible partitions of $\mathcal{V}$.

(1.1) and (1.2) are also referred to as *cut-size*, implying the picture that one takes the whole graph, "cuts" these edges and is left with the subgraphs induced by the subsets of the vertices. Other names used include *edgecut* or *cost* of the partition.

As described above, we are looking for a small subset of edges that, when taken away, separates the graph in two disjoint subsets. This is called finding an *edge separator*.

Instead of edges, one can also look for a subset of vertices, the so-called *vertex separator* that separates the graph:

Given a graph $G = (\mathcal{V}, \mathcal{E})$, find three disjoint subsets $\mathcal{V}_1, \mathcal{V}_2$ and $\mathcal{V}_S$ with

- $\mathcal{V}_1 \dot{\cup} \mathcal{V}_2 \dot{\cup} \mathcal{V}_S = \mathcal{V}$
- $|\mathcal{V}_S|$ small
- $|\mathcal{V}_1| \approx |\mathcal{V}_2|$
- no edge joins $\mathcal{V}_1$ and $\mathcal{V}_2$

Depending on the application, either the edge separator or the vertex separator is needed. Fortunately, it is easy to convert these two kinds of separators into each other.

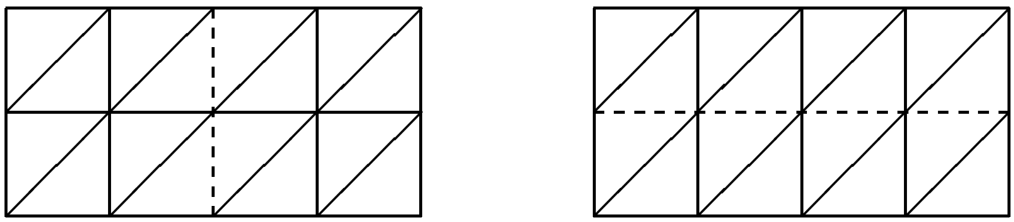
For example, assume we have an edge separator but need a vertex separator. Consider the graph $\bar{\mathcal{G}}$ consisting only of the separating edges and their endpoints. Any vertex cover of $\bar{\mathcal{G}}$ (that is, any set of vertices that include at least one endpoint of every edge in G) is a vertex separator for the graph. Since $\bar{\mathcal{G}}$ is bipartite, we can compute the smallest vertex cover efficiently by bipartite matching.

1.3 Examples

In the following examples we assume that we have a parallel computer with distributed memory, that is, each processor has its own memory. If it needs to access data in some other processors memory, communication is necessary. Many of the parallel computers of today are of this kind.

1.3.1 Partial Differential Equations

A lot of the methods for solving partial differential equations (PDEs) are grid-oriented, i.e. the data is defined on a discrete grid of points, finite elements or finite volumes and the calculation consists of applying certain operations on the data associated with all the points, elements or volumes of the grid [61]. These calculations usually also involve some of the data of the neighboring elements.



good partition   bad partition

Figure 1.1: Two partitions

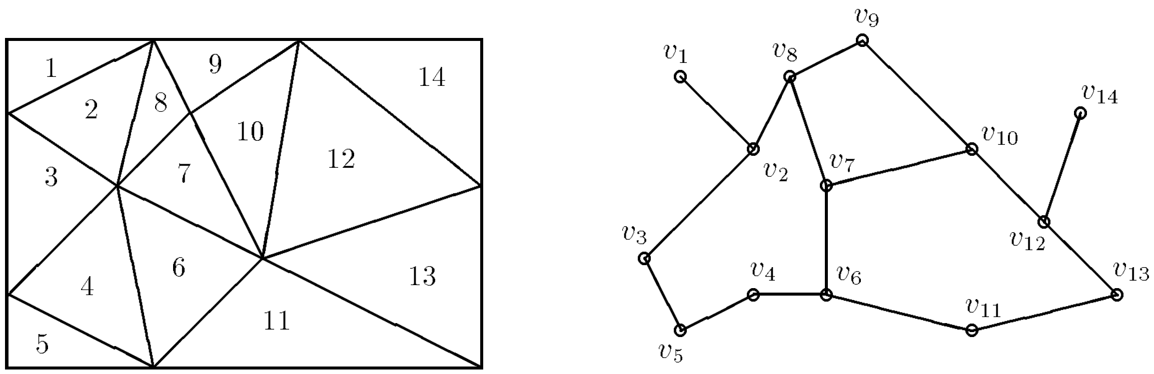


Figure 1.2: A finite element mesh and the corresponding dependency graph

Distributing the problem to parallel processors is done by partitioning the grid into subgrids and solving the associated subproblems on the different processors. Since the elements on the "borders" of the subgrids need to access data on the other processors, communication is necessary. Using the reasonable assumption that the amount of communication between two neighbors is constant, one has to minimize the "border length", i.e. the number of neighbors in different partitions. (cf. Figure 1.1)

To formulate the problem as a graph partitioning problem, we look at the so-called dependency graph of the grid (cf. Figure 1.2).

Every grid-point (or finite element) has a corresponding vertex on this graph. The weight of the vertex is proportional to the amount of computational work done on this grid point. (Quite often the weights of all vertices are equal).

For each pair of neighboring (or otherwise dependent) grid points the corresponding vertices are connected by an edge. The weight of this edge is proportional to the amount of communication between these two grid points. (Again, quite often the weights of all edges are equal).

For finite element meshes (A *mesh* is a graph embedded into (usually) two– or three–dimensional space so that the coordinates of the vertices are known.), the unweighted dependency graph is equal to the *dual graph* of the mesh.

Now the optimal grid partitioning can simply be found by solving the graph partitioning problem on the dependency graph and distributing the gridpoint or finite element to the j-th processor if its corresponding vertex is in the j-th partition of the graph.

1.3.2 Sparse Matrix-Vector Multiplication

A sparse matrix is a matrix containing so many zero entries that savings in space or computational work can be achieved by taking them into consideration in the algorithm.

A fundamental operation in many matrix related algorithms is the multiplication of the matrix A with a vector x. For example, in iterative methods for the solution of linear system (like cg [25], QMR [22] or GMRES [57]), the amount of work spend on this operation can dominate the total amount of work done.

Now we want to distribute the matrix on a parallel computer with p processors. One of the ways that a sparse matrix can be distributed is row-wise, i.e. each row of the matrix is distributed to one processor. If the i-th row of A is kept on a processor then so is the i-th element x_i of the vector x.

How should the rows be distributed amongst the processors for the workload to be balanced and the necessary communication minimized?

Consider the workload: to calculate the i-th element of the vector $y = Ax$, we need to calculate

$$y_i = \sum_{j=1}^{n} a_{ij} x_j.$$

But, since A is sparse, it suffices to calculate

$$y_i = \sum_{j:\, a_{ij} \neq 0} a_{ij} x_j. \tag{1.3}$$

So, the amount of work to calculate y_i is proportional to the amount of non-zero elements in the i-th row of A.

But, as we see in (1.3), we not only need the values of the i-th row of A but also of some of the x_j, namely those where $a_{ij} \neq 0$. Now, if those values are stored on the same processor, we are in luck. Otherwise, we have to communicate with the processor where those values are stored.

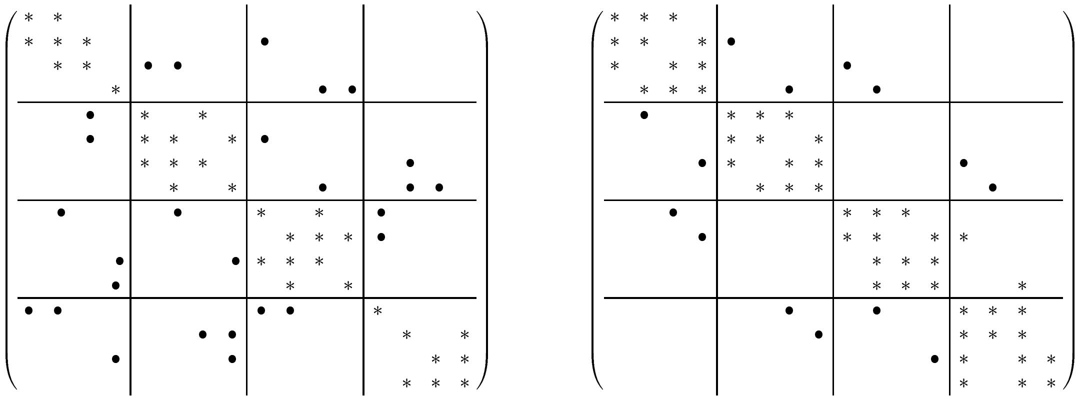


Figure 1.3: Distribution of matrix rows to processors

To minimize communication, we will have to try to store those values (and consequently the associated rows) on the same processor. Thus we want to minimize the occurrence of the following: the i-th row of A is stored on a processor, $a_{ij} \neq 0$ but the j-th row of A is stored on another processor.

Figure 1.3 shows a bad (on the left side) and a good (on the right) distribution of matrix rows to processors. In each matrix, the first four rows are distributed to one processor, the second four to the next and so on. Elements that cause communication are shown by a $\bullet$. Note that there are many more $\bullet$ in the left matrix. But note that the right matrix is just the left matrix with permuted rows and columns and a different row-processor assignment. Permuting rows and columns and the matching processor distribution does not change the amount of work and communication necessary. (It is just easier to see rows 1–4 as belonging together than e.g. rows 1, 2, 5 and 12.) So, the lesser number of $\bullet$ in the right matrix demonstrates how much communication can be saved by the proper distribution.

Again, the problem can be reformulated as a graph partitioning problem. In order to simplify the example, we will look at a structurally symmetric matrix (i.e. if $a_{ij} \neq 0$ then so is $a_{ji} \neq 0$).

The *graph of a matrix* $A \in \mathbb{R}^{n \times n}$ is defined as follows: It consists of n vertices $v_1, \ldots, v_n$. There is an edge between v_i and v_j (with $i \neq j$) if $a_{ij} \neq 0$.

The *degree* of a vertex is the number of direct neighbors it has. For our purpose, we define the weight $w_v(v_i)$ of a vertex v_i as the degree of this vertex plus 1. Then the weight of v_i is equal to the number of non-zero elements of the i-th row of the matrix and therefore proportional to the amount of work to calculate y_i.

The edges all have unit weight. Each edge $e_{i,j}$ symbolizes a data dependency between row i and row j and corresponding vector element x_j and thus a possible communication.

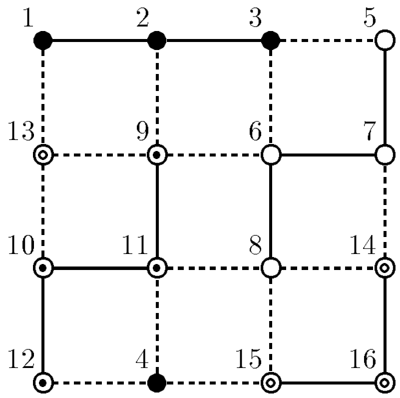


Figure 1.4: Partitioned graphs of matrices in figure 1.3.

Partitioning the weighted graph in p parts with the minimal amount of edges cut allows us to find a good distribution of the sparse matrix on the parallel processors. If v_i is in the j-th subpartition, we store the i-th row of the matrix on processor j.

Since the weight of the vertices is proportional to the work and the partitions are of equal weight, perfect load-balance is achieved. And since each cut edge stands for necessary communication between two processors a minimized cut-size means that communication is minimized.

In Figure 1.4 we show the partitioned graphs of to the matrices in Figure 1.3. The numbers at the vertices correspond to the line/column number in the matrix. Each cut edge corresponds to a $\bullet$ in Figure 1.3. It is now easy to see that both matrices are essentially the same, just with permuted rows/columns leading to a different row-processor assignment.

1.3.3 Other Applications

While both detailed examples above stem from the area of parallelizing mathematical algorithms graph partitioning is applicable to many other, quite different problems and areas as well. One of the first algorithm, the Kernigan–Lin algorithm was developed to assign the components of electronic circuits to circuit boards such that the number of connections between boards is minimized [42]. This algorithm is, with some modifications, still much in use and will be discussed in more detail in section 3.2.1.

Other applications include VLSI (Very Large Scale Integration) design, CAD (Computer Aided Design), decomposition or envelope reduction of sparse matrices, hypertext browsing, geographic information services and physical mapping of DNA (DeoxyriboNucleic Acid, basic constituent of the gene), see, e.g., [38,53] [53] [27,50,52,56] [2][6] [41] [33] and references therein.

1.4 Time vs. Quality

In the sections above we have always talked about "minimizing the cut-size" under certain conditions. But for all but some well-structured or small graphs real minimization is infeasible because it will take to much time. Finding the optimal solution for the graph partitioning problem is for all non-trivial cases known to be NP-complete [23].

Very simply put, this means that there is no known algorithm that is much faster than trying all possible combinations and there is little hope that one will be found.

A little more exact (but still ignoring some fundamentals of the NP theory) it means that graph bisection falls in a huge class of problems all of which can be transformed into each other and for which there are no known algorithms that can solve the problem in polynomial time in the size of the input.

Given a graph $(\mathcal{V}, \mathcal{E})$ without special properties and $k = |\mathcal{V}| + |\mathcal{E}|$ it is suspected that there exists no algorithm for solving the graph bisection problem which runs in $\mathcal{O}(k^n)$ time for *any* given $n \in \mathbb{N}$.

So, instead of finding the optimal solution, we resort to *heuristics*. That is, we try to use algorithms which may not deliver the optimal solution every time but which will give a good solution at least most of the time.

As we will see, there often is a tradeoff between the execution time and the quality of the solution. Some algorithms run quite fast but find only a solution of medium quality while others take a long time but deliver excellent solutions and even others can be tuned between both extremes.

The choice of time vs. quality depends on the intended application. For VLSI-design or network layout it might be acceptable to wait for a very long time because an even slightly better solution can save real money.

On the other hand, in the context of sparse matrix-vector multiplication, we are only interested in the total time. Therefore the execution time for the graph partitioning has to be less than the time saved by the faster matrix-vector multiplication. If we only use a certain matrix once, a fast algorithm delivering only "medium quality" partitions might overall be faster than a slower algorithm with better quality partitions. But if we use the same matrix (or different matrices with the same graph) often, the slower algorithm might well be preferable.

In fact, there are be even more factors to consider. Up to now we wanted the different partitions to be of the same weight. As we will see in section 2.2, it might be of advantage to accept partitions of slightly different size in order to achieve a better cut-size.

All this should demonstrate that there is no single *best* algorithm for all situations and that the different algorithms described in the following chapters all have their applications.

2 Miscellaneous Algorithms

2.1 Introduction

There are many different approaches to graph bisection. Some of them depend on the geometrical properties of the mesh and are thus only applicable for some problems while others only need the graph itself and no additional information. Some take a local view of the graph and only try to improve a given partition while others treat the problem globally. Some are strictly deterministic, always giving the same result while others use quite a lot of random decisions. Some apply graph-theoretic methods while others just treat the problem as a special instance of some other problem (like nonlinear optimization).

In this chapter we will survey some of these algorithms.

The selection of the algorithms in this chapter is somewhat arbitrary but we tried to include both the more commonly used and the ones that are of a certain historical interest. For some other summaries of algorithms, see [32, 53, 61].

In the interest of simplicity, the algorithms will be presented for unweighted graphs wherever generalization to weighted graphs is easy. We also tend to ignore technicalities (like dealing with unconnected graphs) which have to be considered in real implementations of these algorithms.

2.2 Recursive Bisection

As we will see, most of the algorithms are (at least originally) designed for bisection only. But the general formulation of the graph partitioning problem involves finding a p–partition.

It turns out that in many "real-life" applications p is a power of 2, i.e. $p = 2^k$. For example, most parallel computers have 2^k processors. So a natural idea is to work recursively. First partition the graph in two partitions, then partition each of these two in two subpartitions and so on (see Figure 2.1).

But even if we could find a perfect bisection, will this give a partition with (nearly) the same cut-size as a proper p–way partition? As H. Simon and S. Teng have shown in [59], the answer is: No, but ... . In this section, we will recapitulate their re-

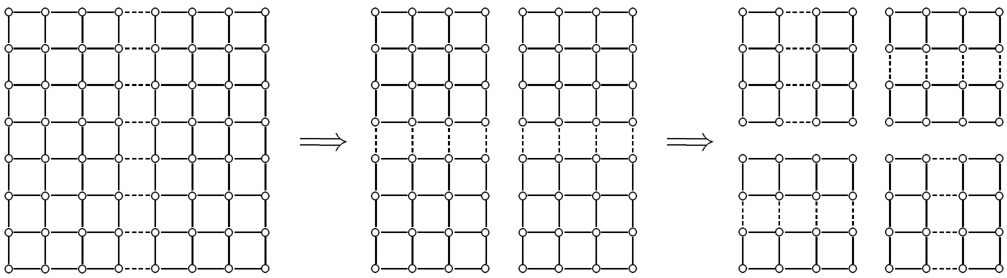


Figure 2.1: Recursive Bisection.

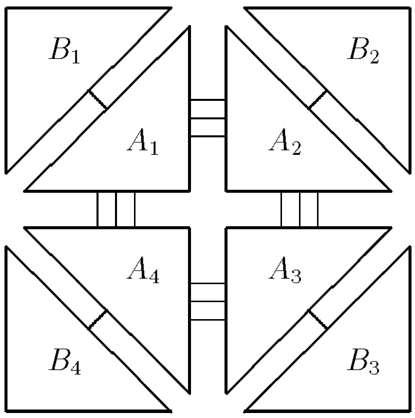


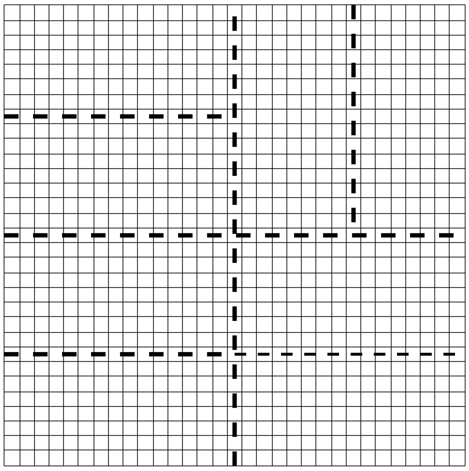
Figure 2.2: Example of a graph with good 4–way partition but bad recursive bisection.

sults: Recursive bisection can deliver nearly arbitrarily bad cut-sizes compared to p–partitions. But that might not be as bad as it sounds. For some important classes of graphs recursive bisection works quite well. And if one does not insist on partitions of exactly equal size, it is possible to use recursive bisection to find partitions that are nearly as good as the ones found by p–way partitioning.

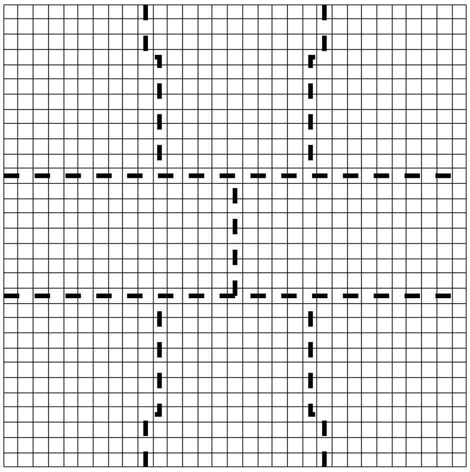
First the bad news: We will demonstrate it by an example: We give a graph with n vertices that has a constant cut-size of 12 using a 4–way partition but where recursive bisection produces a cut-size of size $\mathcal{O}(n^2)$.

Consider the graph sketched in Figure 2.2. Its eight subgraphs A_i, B_i, $(i = 1,2,3,4)$ are cliques (i.e., totally connected graphs) with A_i having $(1/8+\varepsilon_i)n$ vertices and B_i having $(1/8-\varepsilon_i)n$ vertices (with $i = 1,2,3,4$), where the ε_i have the following properties:

1. $-1/8+\delta \leq \varepsilon_i \leq 1/8-\delta$, with $\delta > 0$ fixed and $\varepsilon_i \neq 0$,



a) 8 parts, cut-size 128



b) 8 parts, cut-size 116

Figure 2.3: Real-life Counterexample for recursive bisection

2. $\varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \varepsilon_4 = 0$,
3. $\varepsilon_i + \varepsilon_j \neq 0$ for all $i, j \in \{1,2,3,4\}$,
4. $(1/8 \pm \varepsilon_i)n \in \mathbb{N}$ for $i \in \{1,2,3,4\}$.

(One possible combination would be $\varepsilon_1 = 1/9$, $\varepsilon_2 = 1/11$, $\varepsilon_3 = 1/13$ and $\varepsilon_4 = -(\varepsilon_1 + \varepsilon_2 + \varepsilon_3)$ and $n = 8 * 9 * 11 * 13$.)

The optimal 4–way partition decomposes the graph into $A_i \cup B_i$, $(i = 1,2,3,4)$. The total cut-size is obviously 12.

In contrast, the recursive bisection first decomposes the graph into $\bigcup_{i=1}^{4} A_i$ and $\bigcup_{i=1}^{4} B_i$. But then, in the next recursion level at least one of the A_i and one of the B_i will have to be cut. This is because condition 3 ensures that it is not possible to combine any A_i, A_j (B_i, B_j) to the proper size. Since the A_i and B_j are cliques, cutting even one vertex out of them will increase the cut-size by at least $(1/8 \pm \varepsilon_i)n$ (the number of vertices in A_i resp. B_i). So the cost of the partition is $\mathcal{O}(n^2)$ (at least $2(\delta n)^2 + 4$).

This example can be easily adjusted for general p–partitions. And Figure 2.3 shows that this problem does not only occur in contrived examples. Partitioning a 32×32 grid into 8 parts using (perfect) recursive bisection (Figure 2.3a) will lead to an cut-size of 128 while the optimal 8-partition (Figure 2.3b) has an cut-size of only 116. (Note that in Figure 2.3a, the cuts into 2 and 4 parts are optimal.)

Now for the good news. For some of the more common graphs, it is proved in [59] that recursive bisection is not much worse than real p-partitioning. (In this section, we assume that p–partition and bisection deliver the optimal solution and are not some heuristics.)

A *planar graph* is a graph that can be drawn on a piece of paper so that no edges cross each other. Discretizations of 2–dimensional structures are often planar.

In the finite element method, the domain of the problem is subdivided into a mesh of polyhedral elements. A common choice for an element is a d-dimensional simplex (i.e. a triangle in two dimensions). In classical finite element methods, it is usually a requirement for numerical accuracy that the triangulation is uniformly regular, that is that the simplices are *"well shaped"* [12]. A common shape criteria used in mesh-generation is an upper bound on the aspect ratio of the simplices. This term has many definitions which are roughly equivalent [46]. For the following theorem, the aspect ratio of a simplex T is the radius of the smallest sphere containing T divided by the radius of the largest sphere that can be inscribed in T.

For planar graphs, it is shown in [59, Theorem 4.3] that the solution found by recursive bisection will have a cut-size that is, in the worst case, only $\mathcal{O}(\sqrt{n/p})$ times bigger than that of the p–partition (with $n = |\mathcal{V}|$). With a factor of $\mathcal{O}((n/p)^{1-1/d})$, the same statement holds for well shaped meshes in d dimensions.

Now what happens if we do not insist on partitions of exactly the same size but allow for some imbalance?

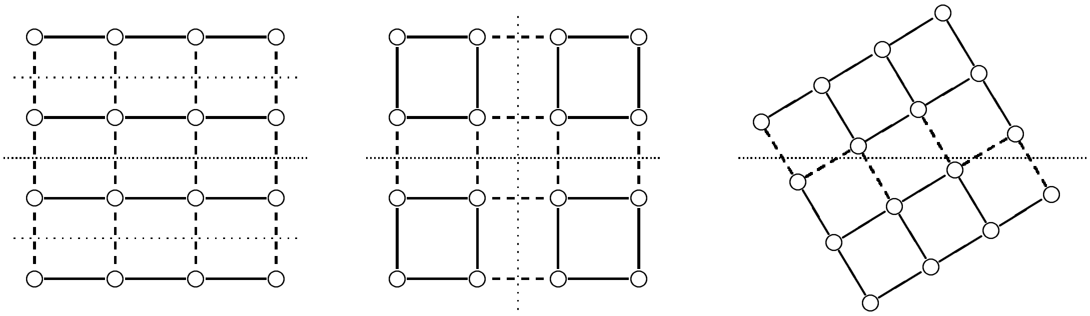
A $(1+\varepsilon, p)$-way partition ($\varepsilon > 0$) is a p–partition where we allow the subpartitions $\mathcal{V}_i$ to be a little bigger than $|\mathcal{V}|/p$, e.g. $|\mathcal{V}_i| \leq (1+\varepsilon)|\mathcal{V}|/p$.

Then Theorem 5.5 in [59] states among other things that if the cut-size of the optimal p–partition is C then one can find a $(1+\varepsilon, p)$–partition with an cut-size of size $\mathcal{O}(C \log p)$ using recursive partitioning. This is a more theoretical result, since the execution time may be exponential in n. But the theorem also shows that there is an algorithm using recursive partitioning that runs in polynomial time and that will find a $(1+\varepsilon, p)$–partition with an cut-size of size of $\mathcal{O}(C \log p \log n)$

This shows that by allowing a little imbalance in the partition size, one can continue to use recursive bisection without paying to much of a penalty in the cut-size compared to real p-partitioning.

2.3 Partitioning with geometric information

Sometimes there is additional information available that may help to solve the problem. For example, in many cases (like in the solution of PDEs, cf. section 1.3.1) the graph is derived from some structure in two- or three-dimensional space, so geometric coordinates can be attached to the vertices and therefore the "geometric layout" of the graph is known. In this case, the graph is also called a *mesh.*



a. b. c.

Figure 2.4: Problems with Coordinate Bisection.

The algorithms described in this section implicitly assume that vertices that are close together physically are close together in the mesh (i.e. connected by a short path). In fact, they make no use of the information about the edges at all. This limits their applicability to problems where such geometric information is both available and meaningful. It also implies that these algorithms cannot easily be extended to graphs with weighted *edges*.

But for these problems, geometrical partitioning techniques are cheap methods that nevertheless produce acceptable partitions.

All of these methods work by finding some structure that divides the underlying space into two parts in such a way that each part contains half of the points (or, for weighted vertices, half of the weight). If some of the points lie *on* the structure, some tie-breaking mechanism will have to be used. Often this structure is a hyperplane (i.e. a plane in two dimensions, a line in two dimensions) but it can also be some other structure like a sphere.

2.3.1 Coordinate Bisection

Coordinate Bisection is the simplest coordinate-based method. It simply consists of finding the hyperplane orthogonal to a chosen coordinate axis (e.g. the y-axis) that divides the points in two equal parts. This is done by only looking at the y-coordinates and finding a value $\bar{y}$ such that half of the points have a y-value of less than $\bar{y}$ and half of the points have a y-value bigger than $\bar{y}$. The separating structure in this case is the hyperplane orthogonal to the y-axis containing the point $(0, \bar{y})$ (resp. $(0, \bar{y}, 0)$).

Recursive application of this technique can be done in two ways.

Always choosing the same axis will lead to thin strips and is usually undesirable, since it leads to long boundaries and thus (probably) to a high cut-size (see Figure 2.4.a).

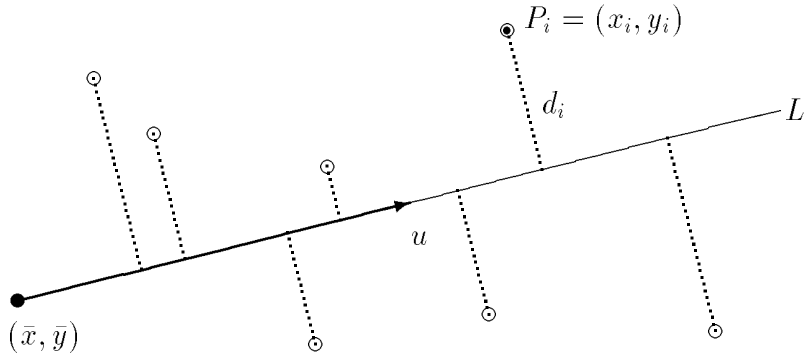


Figure 2.5: Inertial Bisection in 2D.

A better way is to alternately bisect the x-, y- (and z-) coordinates. This leads to a better aspect-ratio and (hopefully) to a lower cut-size (see Figure 2.4.b).

One of the problems of coordinate bisection is that it is coordinate-dependent. Hence in another coordinate system, the same structure may be partitioned quite differently (see Figure 2.4.c). The so-called Inertial Bisection described in the next section tries to remedy this problem by choosing its own coordinate system.

2.3.2 Inertial Bisection

The basic idea of Inertial Bisection (e.g., [32, 53] and references therein) is the following: instead of choosing a hyperplane orthogonal to some fixed coordinate axis one chooses an axis that runs through "the middle" of the points.

Mathematically, we choose the line L in such a way that the sum of the squares of the distance of the mesh points to the line is minimized.

The physical interpretation is that the line is the axis of minimal rotational inertia. If the domain in nearly convex, this axis will align itself with the overall shape of the mesh and the mesh will have a small spatial extend in the directions orthogonal to the axis. This hopefully minimizes the footprint of the domain in the hyperplane and therefore the cut-size.

We state the algorithm in two dimensions. The generalization to three dimensions and to weighted vertices is easy.

Let $P_i = (x_i, y_i)$, $i = 1, \ldots, n$ be the position of the points.

Let the line L be given by a point $\bar{P} = (\bar{x}, \bar{y})$ and a direction $u = (v, w)$ with $\|u\|_2 = \sqrt{v^2 + w^2} = 1$ such that $L = \{\bar{P} + \alpha u \mid \alpha \in \mathbb{R}\}$ (see also Figure 2.5).

Since L is supposed to be the line that minimizes the sum of the squares of the distance from the mesh points to the line it is easy to see that the "center of mass" (i.e. the point that minimizes the sum of the squares of the distance from the mesh points to a single point) lies on L. So we can simply choose

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \qquad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

It remains to choose u so that the following quantity is minimized:

$$\begin{aligned}
&\sum_{i=1}^{n} d_i^2 \\
&= \sum_{i=1}^{n}(x_i - \bar{x})^2 + (y_i - \bar{y})^2 - \big(v(x_i - \bar{x}) + w(y_i - \bar{y})\big)^2 \\
&= (1 - v^2)\sum_{i=1}^{n}(x_i - \bar{x})^2 + (1 - w^2)\sum_{i=1}^{n}(y_i - \bar{y})^2 + 2vw\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \\
&= (1 - v^2)S_{xx} + (1 - w^2)S_{yy} + 2wvS_{xy} \\
&= w^2 S_{xx} + v^2 S_{yy} + 2wvS_{xy} \\
&= u^T \begin{pmatrix} S_{yy} & S_{xy} \\ S_{xy} & S_{xx} \end{pmatrix} u \\
&=: u^T M u
\end{aligned}$$

where S_{yy}, S_{yy} and S_{xy} are the summations in the previous line.

M is symmetric and hence we can choose u to be the normalized eigenvector corresponding to the smallest eigenvalue M in order to minimize this expression (This follows from, e.g., [25, Theorem 8.1.2]).

2.3.3 Geometric Partitioning

Another, rather more complicated method to partition meshes was introduced in [47]. In its original form and for certain graphs it will find a vertex separator that (with a high probability) has an "asymptotically optimal" size and divides the graph in such a way that the sizes of the two partitions are not too different.

Definition 2.1 A *k-ply neighborhood system in d dimensions* is a set $\{D_1, D_2, \ldots, D_n\}$ of closed disks in $\mathbb{R}^d$ such that no point of $\mathbb{R}^d$ is strictly interior to more than k disks.

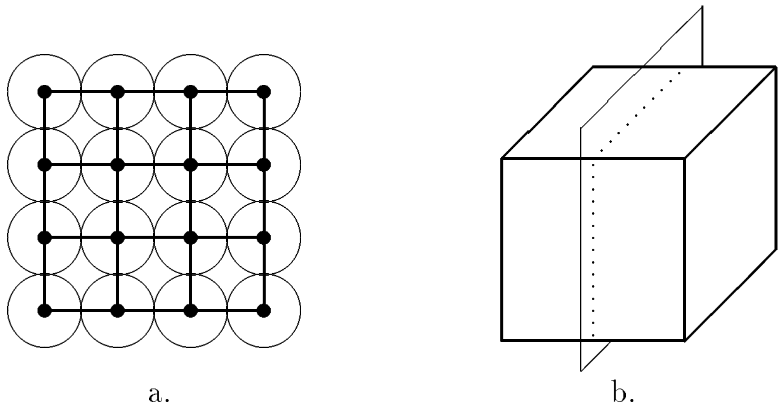


Figure 2.6: a. A regular mesh as a $(1,1)$ overlap graph.
b. Bisecting a 3 dim. cube.

Definition 2.2 An (α, k) *overlap graph* is a graph defined in terms of a k-ply neighborhood system $\{D_1, D_2, \ldots, D_n\}$ and a constant $\alpha \geq 1$.
There is a vertex for each disk D_i. The vertices i and j are connected by an edge if expanding the radius of the smaller of D_i and D_j by a factor α causes the disks to overlap, i.e.

$$\mathcal{E} = \left\{ e_{i,j} \,\middle|\, D_i \cap \alpha D_j \neq 0 \ \text{ and } \ \alpha D_i \cap D_j \neq 0 \right\}$$

Overlap graphs are good models of computational meshes because every well shaped mesh in two or three dimensions and every planar graph are contained in some (α, k)-overlap graph (for suitable α and k).

For example, a regular n-by-n mesh is a $(1,1)$-overlap graph (see Figure 2.6.a).

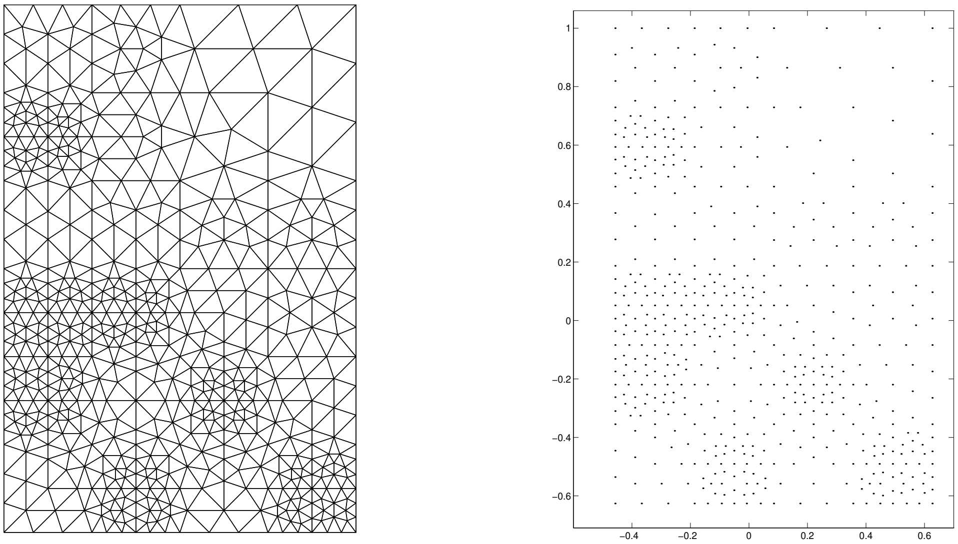
We have the following theorem:

Theorem 2.1 ([47])
Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an (α, k)-overlap graph in d dimensions with n nodes. Then there is a vertex separator $\mathcal{V}_S$ so that $\mathcal{V} = \mathcal{V}_1 \dot{\cup} \mathcal{V}_S \dot{\cup} \mathcal{V}_2$ with the following properties:

1. *$\mathcal{V}_1$ and $\mathcal{V}_2$ each have at most $n(d+1)/(d+2)$ vertices, and*

2. *$\mathcal{V}_S$ has at most $\mathcal{O}(\alpha k^{1/d} n^{(d-1/d)})$ vertices.*

It is easy to see that for regular meshes in simple geometric shapes (like balls, squares or cubes, cf. Figure 2.6.b), the separator size given in the theorem is "asymptotically optimal".



a) The mesh. b) The scaled points.

Figure 2.7: Geometric Bisection I

The proof for this theorem is rather long and involved but it leads to a randomized algorithm running in linear time that will find, with high probability, a separator of the size given in the theorem.

The separator is defined by a d-dimensional sphere. The algorithm randomly chooses the separating circle from a distribution that is constructed so that the separator will satisfy the conclusions of the theorem with high probability.

To describe the algorithm, we will introduce some terminology:

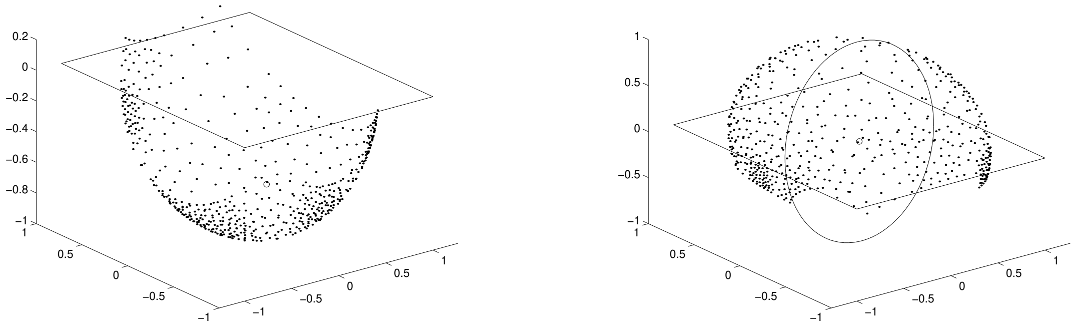
A *stereographic projection* is a mapping of points in $\mathbb{R}^d$ to the unit-sphere centered at the origin in $\mathbb{R}^{d+1}$. First, a point $p \in \mathbb{R}^d$ is embedded in $\mathbb{R}^{d+1}$ by setting the $d+1$ coordinate to 0. It is then projected on the surface of the sphere along a line passing through p and the "north pole" $(0, \ldots, 0, 1)$.

The *centerpoint* of a given set of points in $\mathbb{R}^d$ has the property that every hyperplane through the centerpoint divides the set into two subsets whose sizes differ by at most a ratio of 1:d. Note that the centerpoint does not have to be one of the original points.

With these definitions, the algorithm is as follows (see also Fig. 2.7 – 2.9):

Algorithm 2.1 (Geometric Bisection)

Project up: Stereographically project the vertices from $\mathbb{R}^d$ to the unit-sphere in $\mathbb{R}^{d+1}$.



a) Projected meshpoints with centerpoint.

b) Comformally mapped points with great circle.

Figure 2.8: Geometric Bisection II

Find centerpoint z of the projected points.

Conformal Map: Conformally map the points on the sphere such that the centerpoint now lies at the origin. This is done in the following way: First, rotate the sphere around the origin so that the centerpoint has the coordinates $(0, \dots 0, r)$ for some r. Second, dilate the points so that the (new) centerpoint lies in the origin. This can be done by projecting the rotated points back on the $\mathbb{R}^d$–plane (using the inverse of the stereographic projection), scaling the points in $\mathbb{R}^d$ by a factor of $\sqrt{(1-r)/(1+r)}$, and then stereographically projecting the points back on the sphere.

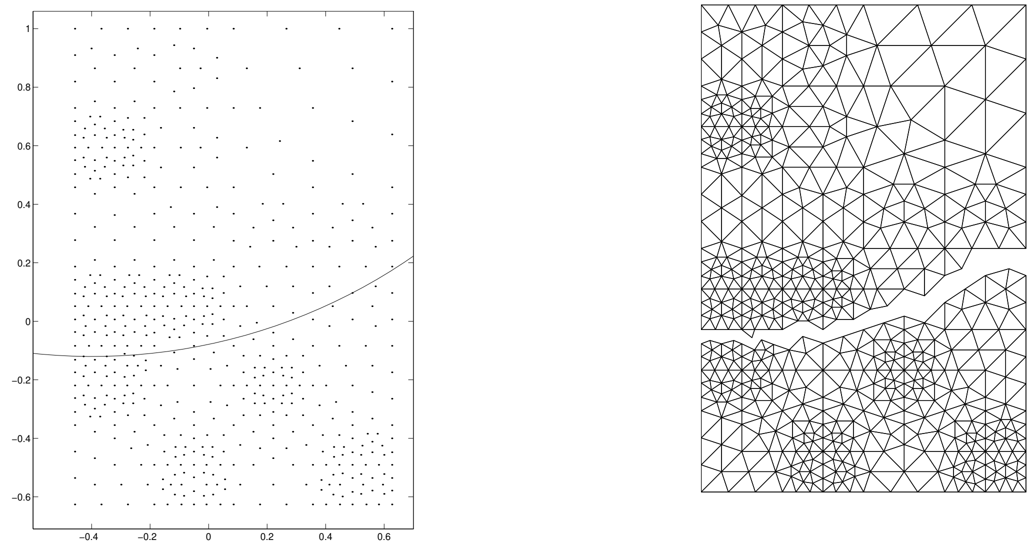
Find a great circle by intersecting the $\mathbb{R}^{d+1}$–sphere with a random d–dimensional hyperplane through the origin.

Unmap the great circle to a circle C in $\mathbb{R}^d$ by inverting the dilation, the rotation and the stereographic projection.

Find separator: A *vertex* separator is found by choosing all those vertices whose corresponding disk (cf. Def. 2.2), magnified by a factor of α intersects C.

While implementing this algorithm, the following simplifications (suggested in [24]) can be made.

Instead of finding a vertex separator, it is easier to find an edge-separator by choosing all those edges cut by the circle C. This has the advantage that one does not have to know the neighborhood system for the graph.



a) Separating circle projected back on the plane.

b) The edge separator induced by the separating circle.

Figure 2.9: Geometric Bisection III

The theorem only guarantees a ratio of $1 : d+1$ between the two partitions while for bisection we want the two partitions to be of the same size (possibly up to a difference of one vertex). To achieve this, the hyperplane is moved along its normal vector until it divides the points evenly. Thus the separator is a circle, but not a great circle, on the unit-sphere in $\mathbb{R}^{d+1}$; its projection to the $\mathbb{R}^d$ plane is still a circle (or possibly a line).

The centerpoint can be found by linear programming and although this is done in polynomial time it is still a slow process. So, instead of using the real centerpoint, one can use a heuristic to find an approximate centerpoint. This can be done in linear time using a randomized algorithm.

To speed up the computation even more, only a randomly chosen subset of points is used to compute the centerpoint.

A random great circle has a good possibility to induce a good partition. But according to [24] experiments showed that it is worthwhile to generate different circles and choose the one that delivers the best result. One can further improve the algorithm by modifying this selection such that the normal vector of the randomly generated great circles are biased in the direction of the moment of inertia of the points (see also Section 2.3.2: Inertial Bisection).

3 Partitioning without geometric information

3.1 Introduction

All of the methods mentioned above work quite well on some meshes but they have a couple of disadvantages. First, they assume that connected vertices are in some sense geometrically adjacent. While this is certainly the case for some applications, it does not hold in general. Secondly, these methods require geometric information about the mesh points. Even in cases where the first condition is satisfied, the layout of the mesh may not be available.

Consequently, methods that do not use geometric information but only consider the connectivity information of the graph are applicable for a wider range of problems.

3.2 Graph Growing and Greedy Algorithms

When thinking about partitioning algorithms, one obvious idea is the following: Choose one starting vertex and in some way add vertices to it until the partition is big enough. These algorithms can be classified as *"greedy–"* and *"graph-growing–"*type algorithms.

In greedy-algorithms (e.g., [11]), the next vertex added is one that is "best" in some sense (e.g., it increases the cut-size by the least amount) while in graph-growing algorithms, the vertices are added in such a way that the resulting subgraph grows in a certain way (e.g., breadth–first). Other graph-growing algorithms may for example start with several starting vertices at the same time and grow the subgraphs in parallel or may have some preference to grow in a certain direction.

Since the above describes a whole family of algorithms, we will describe some examples a bit more closely.

First consider the following greedy approach:

Algorithm 3.1 (Greedy Partitioning)

Unmark all vertices.

Choose a pseudo-peripheral vertex (one of a pair of vertices that are approximately at the greatest distance from each other in the graph, cf. [58]) as starting vertex, mark it and add it to the first partition.

For the desired number of partitions **do**

- **repeat**
  - Among all the unmarked vertices adjacent to the current partition, choose the one with the least number of unmarked neighbors.
  - Mark it and add it to the current partition.

  until the current partition is big enough.
- If there are unmarked vertices left, choose one adjacent to the current partition with the least number of unmarked neighbors as starting vertex for the next partition.

Note that by choosing new vertices only adjacent to the current partition, this algorithm tries to keep the subpartitions connected and thus also does some kind of graph growing.

Another well-known algorithm that is often called "greedy" is the Farhat–algorithm ([17]). A closer look reveals that it is a graph growing algorithm which only chooses the starting vertices of each partition in a greedy way.

Adapted to work on a graph (the original works on nodes and elements of a FEM mesh) it can be described as follows:

Algorithm 3.2 (Farhat's Algorithm)

Unmark all vertices.

For the desired number of partitions **do**

- Among all the vertices chosen for the last partition, choose the one with the smallest nonzero number of unmarked neighbors, mark all these neighbors and add them to the current partition. (For the first vertex, one could again choose a pseudo-peripheral vertex, mark it and add it to the current partition.)
- **repeat**

  For each vertex v in the current partition **do**
  - **For** each unmarked neighbor v_n of v **do**
    - · Add v_n to the current partition.
    - · mark v_n.

  until the current partition is big enough.

These algorithms, in particular the graph growing ones, are very fast.They also have the additional advantage of being able to divide the graph into the desired number of partitions directly, thereby avoiding recursive bisection. Therefore their running time is essentially independent of the desired number of subpartitions.

The ability to divide the graph into any number of partitions is also attractive if the desired number is not a power of 2.

On the downside, the quality of the partitions (measured in cut-size) is not always great and when partitioning complicated graphs into several subpartitions the last subpartitions tend to be disconnected (see e.g., [61]). They also tend to be sensitive to the choice of the starting vertex (but since they are quite fast, one can run the algorithms with different starting vertices and simply choose the best result).

3.2.1 Kernigan–Lin Algorithm

The Kernigan–Lin algorithm [42], often abbreviated as K/L, is one of the earliest graph partitioning algorithms and was originally developed to optimize the placement of electronic circuits onto printed circuit cards so as to minimize the number of connections between cards.

The K/L algorithm does not create partitions but rather improves them iteratively (there is no need for the partitions to be of equal size). The original idea was to take a random partition and apply Kernigan–Lin to it. This would be repeated several times and the best result chosen. While for small graphs this delivers reasonable results, it is quite inefficient for larger problem sizes.

Nowadays, the algorithm is used to improve partitions found by other algorithms. As we will see, K/L has a somewhat "local" view of the problem, trying to improve partitions by exchanging neighboring nodes. So it nicely complements algorithms which have a more "global" view of the problem but tend to ignore local characteristics. Examples for these kinds of algorithms would be inertial partitioning and spectral partitioning (see sections 2.3.2 and 3.3 resp.).

Important practical advances were made by C. Fiduccia and R. Mattheyses in [18] who implemented the algorithm in such a way that one iteration runs in $\mathcal{O}(|\mathcal{E}|)$ instead of $\mathcal{O}(|\mathcal{V}|^2 \log |\mathcal{V}|)$. We will first describe the original algorithm and later discuss these and other improvements.

Let us introduce some notation: It is actually easier to describe the algorithm for a graph with weighted edges; so let $(\mathcal{V}, \mathcal{E}, W_{\mathcal{E}})$ be a graph with given subpartitions $\mathcal{V}_A \dot{\cup} \mathcal{V}_B = \mathcal{V}$.

The *diff–value* of a vertex v is the amount the cut-size will decrease if it is moved to the other partition, i.e., for $v \in \mathcal{V}_A$:

$$\operatorname{diff}(v) = \operatorname{diff}(v, \mathcal{V}_A, \mathcal{V}_B) := \sum_{b \in \mathcal{V}_B} w_e(e_{v,b}) - \sum_{a \in \mathcal{V}_A} w_e(e_{v,a})$$

Note that if a vertex is moved from one subpartition to the other, only its diff–value and the diff–values of its neighbors change.

The *gain–value* of a pair $a \in \mathcal{V}_A, b \in \mathcal{V}_B$ of vertices is the amount the cut-size changes if we swap a into subpartition $\mathcal{V}_B$ and b into $\mathcal{V}_A$. One can easily see that

$$\operatorname{gain}(a, b) = \operatorname{gain}(a, b, \mathcal{V}_A, \mathcal{V}_B) := \operatorname{diff}(a) + \operatorname{diff}(b) - 2w_e(e_{a,b}).$$

With this, we will now describe one pass of the algorithm:

Algorithm 3.3 (Kernigan–Lin)

Given two subpartitions $\mathcal{V}_A, \mathcal{V}_B$

Compute the diff–value of all vertices.

Unmark all vertices.

Let k_0 = cut-size.

For $i = 1$ **to** $\min(|\mathcal{V}_A|, |\mathcal{V}_B|)$ **do**

- Among all unmarked vertices, find the pair (a_i, b_i) with the biggest gain (which might be negative).
- mark a_i and b_i.
- **For** each neighbor v of a_i or b_i **do**

  Update diff(v) as if a_i and b_i had been swapped, i.e.,

$$\operatorname{diff}(v) := \operatorname{diff}(v) + \begin{cases} 2w_e(e_{v,a_i}) - 2w_e(e_{v,b_i}) & \text{for } v \in \mathcal{V}_A \\ 2w_e(e_{v,b_i}) - 2w_e(e_{v,a_i}) & \text{for } v \in \mathcal{V}_B \end{cases}$$

- $k_i = k_{i-1} + \operatorname{gain}(a_i, b_i)$, i.e., k_i would be the cut-size if $a_1, a_2, \ldots, a_i$ and $b_1, b_2, \ldots, b_i$ had been swapped.

Find the smallest j such that $k_j = \min_i k_i$.

Swap the first j pairs, i.e.,

$$\mathcal{V}_A := \mathcal{V}_A - \{a_1, a_2, \ldots, a_j\} \cup \{b_1, b_2, \ldots, b_j\}$$
$$\mathcal{V}_B := \mathcal{V}_B - \{b_1, b_2, \ldots, b_j\} \cup \{a_1, a_2, \ldots, b_j\}$$

Continue these iterations until no further cut-size improvement is achieved.

So, in each phase, K/L swaps pairs of vertices chosen to maximize the gain. It continues doing so until all vertices of the smaller subpartition are swapped. The important part is that the algorithm does *not* stop as soon as there are no more improvements to be made. Instead it continues, even accepting negative gains in the hope that later gains will be large and that the overall cut-size will later be reduced. This ability to climb out of local minima is a crucial feature. (See also Figure 3.1, taken from [15]).

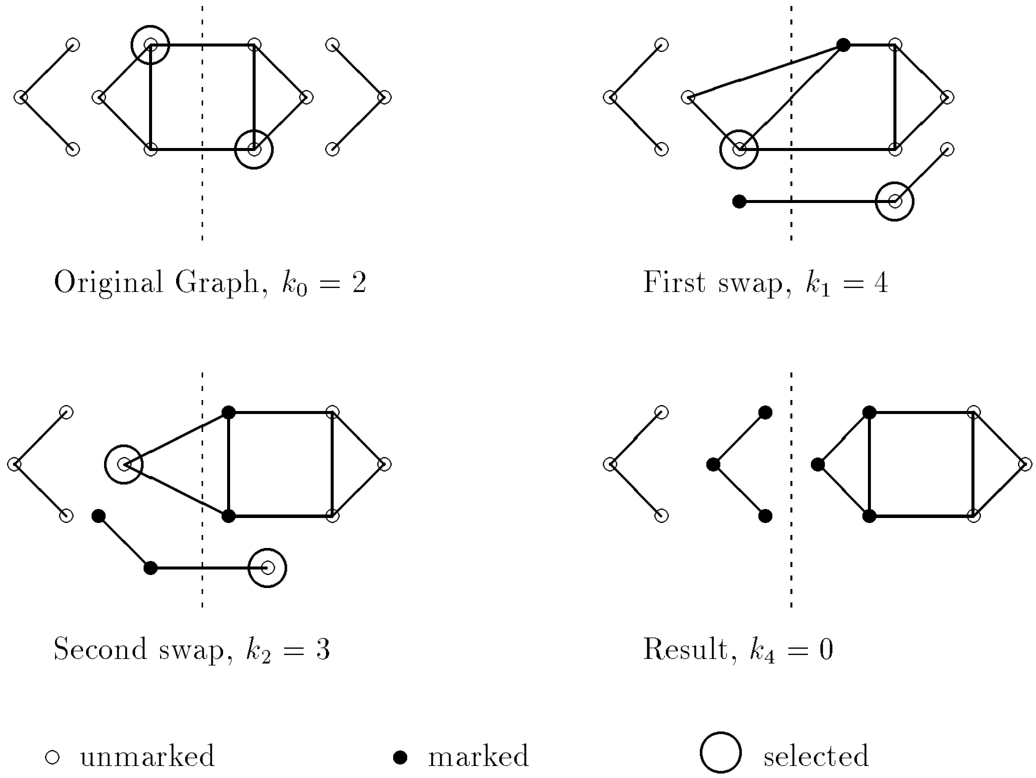


Figure 3.1: The Kernigan–Lin algorithm at work

As mentioned above, the algorithm implemented in [18] takes $\mathcal{O}(|\mathcal{E}|)$ time for one iteration. The reduction is partly achieved by choosing single nodes to be swapped instead of pairs. Furthermore, the diff–values of the vertices are bucket-sorted and the moves associated with each gain value are stored in a doubly linked list. Choosing a move with the highest diff–value involves finding a nonempty list with the highest diff–value, while updating the diff–value of a vertex is done by deleting it from one doubly linked list and inserting it into another (this can be done in constant time).

There are many variations of the Kernigan–Lin algorithm ([31, 39, 44, 54, 67]) often trading execution time against quality or generalizing the algorithm to more than two subpartitions. Some examples are:

- Instead of exchanging all vertices, limit the exchange to some fixed number or allow only a fixed number of exchanges with negative gain. This is based on the observation that often the largest gain is achieved early on.

- Use only a fixed number of inner iterations (often only one).

- Only evaluate diff–values of vertices near the boundary since it is unlikely that moving inner nodes will be helpful.

3.3 Spectral Bisection

The *spectral bisection* algorithm described in this section is quite different from all the other algorithms described so far. It does not use geometric information yet it does not operate on the graph itself but on some mathematical representation of it. While the other algorithms use (almost) no floating point calculations at all, the main computational effort of the spectral bisection algorithm are standard vector operations on floating point numbers. The decision about each vertex is based on the whole graph, so it takes a more global view of the problem.

The method itself is quite easy to explain but why the heuristic works is not at all obvious.

DEFINITION 3.1 Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with $n = |\mathcal{V}|$ vertices numbered in some way and with $\deg(v_i)$ the degree of vertex i (i.e., the number of adjacent vertices). The *Laplacian matrix* $L(\mathcal{G})$ (or only L if the context is obvious) is an $n \times n$ symmetric matrix with one row and column for each vertex. Its entries are defined as follows:

$$l_{ij} = \begin{cases} -1 & \text{if } i \neq j \text{ and } e_{i,j} \in \mathcal{E} \\ 0 & \text{if } i \neq j \text{ and } e_{i,j} \notin \mathcal{E} \\ \deg(v_i) & \text{if } i = j \end{cases}$$

The same matrix with a zero diagonal is often called the *adjacency matrix* $A(\mathcal{G})$ of $\mathcal{G}$.

Since a permutation of the vertex numbering will lead to a different matrix it is sloppy to speak of *the* Laplacian of a graph. But since matrices induced by different vertex numberings can be derived from each other by permuting rows and columns they share most properties, so we will continue to speak of *the* Laplacian unless a closer distinction is necessary.

The Laplacian has some nice properties (cf. e.g., [19,45]):

THEOREM 3.1

1. *$L(\mathcal{G})$ is real and symmetric, its eigenvalues $\lambda_1 \le \lambda_1 \le \cdots \le \lambda_n$ are therefore real and its eigenvectors are real and orthogonal.*
2. *All eigenvalues λ_j are nonnegative and hence $L(\mathcal{G})$ is positive semidefinite.*
3. *With $e := (1, 1, \ldots, 1)^T$, $L(\mathcal{G})e = 0 \cdot e$, so $\lambda_1 = 0$ with e an associated eigenvector.*
4. *The multiplicity of the zero eigenvalue is equal to the number of connected components of $\mathcal{G}$. In particular:*

$$\lambda_2 \neq 0 \;\Leftrightarrow\; \mathcal{G} \text{ is connected.} \tag{3.1}$$

Proof. 1. and 3. are trivial and 2. follows from the Gershgorin Circle Theorem ([25, Theorem 7.2.1]) together with the simple fact (from the definition of L) that

$$\sum_{\substack{i=1 \\ i \neq j}}^{n} |l_{ij}| = l_{jj} > 0 \quad \text{for all } j \in \{1, \ldots, n\}$$

and so all Gershgorin circles have nonnegative real parts. Since all eigenvalues are real, they are all nonnegative. This also shows that $\lambda_n \le 2\max_i(\deg(v_i)) \le 2(n-1)$.

For part 4, we first show that $\lambda_2 \neq 0$ if $\mathcal{G}$ is connected. Note that $\mathcal{G}$ is connected if and only if $L(\mathcal{G})$ is irreducible and if L is irreducible then so is the nonnegative matrix $\tilde{L} = 2nI - L$. Also, λ_i is an eigenvalue of L if and only if $\tilde{\lambda}_{n-i} = 2n - \lambda_i$ is an eigenvalue of $\tilde{L}$. In particular, the largest eigenvalue $\tilde{\lambda}_n$ of $\tilde{L}$ and all other eigenvalues of $\tilde{L}$ are positive.

The Perron–Frobenius theorem ([5, Theorem 2.1.4]) applied to $\tilde{L}$ implies that if $\tilde{L}$ is connected, $\tilde{\lambda}_n$ is a simple eigenvalue and hence so is λ_1. Therefore $\lambda_2 > \lambda_1 = 0$.

Now assume that $\mathcal{G}$ not connected but that it consists of k connected components $\mathcal{G}_1, \ldots, \mathcal{G}_k$. By first numbering the vertices of $\mathcal{G}_1$, then of $\mathcal{G}_2$ and so on, $L(\mathcal{G})$ has block-diagonal form with $L(\mathcal{G}_1), \ldots, L(\mathcal{G}_k)$ as diagonal blocks. So $\sigma(L(\mathcal{G})) = \bigcup_{i=1}^{k} \sigma(L(\mathcal{G}_i))$ and since each of the $\sigma(L(\mathcal{G}_i))$ contains one 0, the first k eigenvalues of $L(\mathcal{G})$ are 0.

We have thus shown that $\mathcal{G}$ is connected if and only if $\lambda_2 \neq 0$.

But what if $\lambda_1 = \lambda_2 = \cdots = \lambda_k = 0$? Then $\mathcal{G}$ cannot be connected and consequently consists of at least two unconnected components $\mathcal{G}_1$ and $\mathcal{G}_2$. As above we see that $\sigma(L(\mathcal{G})) = \sigma(L(\mathcal{G}_1)) \cup \sigma(L(\mathcal{G}_2))$ and if $k > 2$, at least one of the submatrices has more than one zero-eigenvalue and is therefore disconnected. Repeating this argument, we see that $\mathcal{G}$ has k connected components. □

Equation (3.1) also motivates the following:

DEFINITION 3.2 $\lambda_2(L(\mathcal{G}))$ is called the *algebraic connectivity.*

The corresponding eigenvector is often called *Fiedler vector*. It was named to honor Miroslav Fiedler's pioneering work on these objects [19, 20]. One of his discoveries, which serves as a first (very) small motivation for the following algorithm, is the following

Theorem 3.2 ([20])
Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (with $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$) be a connected graph and let u be its Fiedler vector. For any given real number $r \geq 0$, define $\mathcal{V}_1 := \{v_i \in \mathcal{V} \mid u_i \geq -r\}$. Then the subgraph induced by $\mathcal{V}_1$ is connected.
Similarly, for a real number $r \leq 0$, the subgraph induced by $\mathcal{V}_2 := \{v_i \in \mathcal{V} \mid u_i \leq -r\}$ is also connected.

Note that the theorem is independent of the length and sign of the Fiedler vector.

This theorem does not imply anything about a good bisection but it does imply that the Fiedler vector does somehow divide the vertices in a "sensible" way. After all, it guarantees at least one connected partition. This is much more than what wo would achieve by choosing just a random subset.

This leads us to the idea of using the Fiedler vector to bisect the graph. We will first describe the algorithm and then later try to explain why it works.

Algorithm 3.4 (Spectral Bisection)

Given a connected graph $\mathcal{G}$, number the vertices in some way and form the Laplacian matrix $L(\mathcal{G})$.

Calculate the second-smallest eigenvalue λ_2 and its eigenvector u.

Calculate the median $\mathsf{m_u}$ of all the components of u.

Choose $\mathcal{V}_1 := \{\mathsf{v_i} \in \mathcal{V} \mid \mathsf{u_i} < \mathsf{m_u}\}$, $\mathcal{V}_2 := \{\mathsf{v_i} \in \mathcal{V} \mid \mathsf{u_i} > \mathsf{m_u}\}$, and, if some elements of u equal $\mathsf{m_u}$, distribute the corresponding vertices so that the partition is balanced.

But why does this algorithm work? There are some very different approaches that try to answer this question. In addition to the graph-theoretic approach (cf. Theorem 3.2), one can look at graph bisection as a discrete optimization problem (in [3, 29]) or as a quadratic assignment problem (in [53]), look at several physical models (like a vibrating string or a net of connected weights) (in [30, 51]) or examine the output of a certain neural net trained to bisect a graph (in [21]).

Among all of these, the following approach via the discrete optimization formulation is (in the authors opinion) the most convincing derivation of the spectral bisection algorithm.

We have a graph $\mathcal{G} = (\mathcal{V},\mathcal{E})$ (with $n := |\mathcal{V}|$ even) which we want to partition into two equal sized parts $\mathcal{V}_1$ and $\mathcal{V}_2$. To designate which vertex belongs to which part, we use an index vector $x \in \{\pm 1\}^n$ such that

$$x_i = \begin{cases} 1 & \text{if } v_i \in \mathcal{V}_1 \\ -1 & \text{if } v_i \in \mathcal{V}_2 \end{cases} \tag{3.2}$$

Notice that the function

$$f(x) := \frac{1}{4} \sum_{(i,j)\in\mathcal{E}} (x_i - x_j)^2$$

denotes the number of cut edges since $x_i - x_j = 0$ if both v_i and v_j lie in the same subpartition. On the other hand, if v_i and v_j lie in different partitions, they have a different sign and thus $(x_i - x_j)^2 = 4$.

We rewrite

$$\sum_{(i,j)\in\mathcal{E}} (x_i - x_j)^2 = \sum_{(i,j)\in\mathcal{E}} (x_i^2 + x_j^2) - \sum_{(i,j)\in\mathcal{E}} 2x_i x_j$$

and note that

$$-\sum_{(i,j)\in\mathcal{E}} 2x_i x_j = \sum_{v_i\in\mathcal{V}} \sum_{v_j\in\mathcal{V}} x_i A_{ij} x_j = x^T A x$$

with A the adjacency matrix and

$$\sum_{(i,j)\in\mathcal{E}} (x_i^2 + x_j^2) = \sum_{(i,j)\in\mathcal{E}} 2 = 2|\mathcal{E}| = \sum_{i=1}^{n} \deg(v_i) = x^T D\, x$$

with $D = L - A$ a diagonal matrix with the vertex degrees on the diagonal, i.e., $D = \mathrm{diag}_i(\deg(v_i))$ and so (cf. Definition 3.1)

$$f(x) = \tfrac{1}{4} x^T L x \tag{3.3}$$

Now the graph bisection problem can be formulated as:

$$\begin{array}{ll} \text{Minimize} & f(x) = \tfrac{1}{4} x^T L x \\ \text{subject to} & x \in \{\pm 1\}^n \\ & x^T e = 0 \end{array} \tag{3.4}$$

where again $e = (1, 1, \ldots, 1)$. The third condition, $x^T e = 0$ forces the sets to be of equal size.

Graph partitioning is NP-complete and (3.4) is just another formulation, so we cannot really solve this problem either. But we can relax the constraints a bit: instead of choosing discrete values for the x_i we look for a real vector z which has the same (Euclidean) length as x and fulfills the other conditions:

$$\begin{array}{ll} \text{Minimize} & f(z) = \frac{1}{4} z^T L z \\ \text{subject to} & z^T z = n \\ & z^T e = 0 \end{array} \tag{3.5}$$

As we will see, this problem is easy to solve. Since the feasible set of the continuous problem (3.4) is a subset of the feasible set of the discrete problem (3.5), every solution of (3.5) will give us a lower bound for (3.4) and we hope that the continuous solution will give us a good approximation to the solution of (3.4).

From Theorem 3.1 we know that the eigenvectors $u_1, u_2, \ldots, u_n$ of L are orthonormal (and therefore span $\mathbb{R}^n$) and that $u_1 = \sqrt{n}e$. We can thus decompose every $z \in \mathbb{R}^n$ into a linear combination $z = \sum_{i=1}^n \alpha_i u_i$ of these vectors.

In order to to meet the restrictions, we have to place some conditions on the α_i. We want $z^T e = 0$ and since

$$z^T e = \Big(\sum_{i=1}^n \alpha_i u_i\Big)^T u_1 = \sum_{i=1}^n (\alpha_i u_i)^T u_1 = \alpha_1 \, u_1^T u_1 = \alpha_1$$

we need to have $\alpha_1 = 0$. For $z^T z = n$, we need

$$\begin{aligned} z^T z &= \Big(\sum_{i=1}^n \alpha_i u_i\Big)^T \Big(\sum_{i=1}^n \alpha_i u_i\Big) \\ &= \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j u_i^T u_j = \sum_{i=1}^n \alpha_i^2 = n \end{aligned}$$

Furthermore, we have

$$\begin{aligned} 4f(z) &= z^T L z = \Big(\sum_{i=1}^n \alpha_i u_i\Big)^T L \, \Big(\sum_{i=1}^n \alpha_i u_i\Big) \\ &= \Big(\sum_{i=1}^n \alpha_i u_i\Big)^T \Big(\sum_{i=1}^n \alpha_i \lambda_i u_i\Big) = \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j \lambda_j u_i^T u_j \\ &= \sum_{i=1}^n \alpha_i^2 \lambda_i \\ &= \sum_{i=2}^n \alpha_i^2 \lambda_i \geq \lambda_2 \sum_{i=1}^n \alpha_i^2 = \lambda_2 n \end{aligned} \tag{3.6}$$

where (3.6) holds since $\alpha_1 = 0$ and the inequality follows from $0 < \lambda_2 \leq \lambda_i$ for $i = 3, \ldots, n$. Since we can achieve $4f(z) = \lambda_2 n$ by choosing $z = \sqrt{n} u_2$, we find that the correctly scaled Fiedler vector $\sqrt{n} u_2$ (which satisfies both constraints) is a solution of the continuous optimization problem (3.5). In fact, if $\lambda_3 > \lambda_2$ the solution is unique up to a factor of -1(cf. also [35, p. 178]). In addition, $\frac{n}{4}\lambda_2$ is a lower bound for the cut-size of a bisection of $\mathcal{V}$.

Having found the solution of the continuous problem (3.5), we need to map it back to a discrete partition and hope that the result is a good approximation to the solution of the discrete problem (3.4).

There seem to be two "natural" ways to do this. The first method ([10]) places more importance on the sign of the result and just map $x_i = \operatorname{sign}(z_i)$, dealing with zero elements of x in some way (e.g., assigning them all to one partition). Its drawback is that the partitions need not be balanced. Consequently, some additional work is necessary.

The second (and more often used) method (e.g., [29, 51, 61]) places its emphasis on the balance and assigns the largest $n/2$ elements of z to one partition and the rest to the other partition. This can easily be done by computing the median $\bar{z}$ of the z_i, setting

$$x_i = \begin{cases} -1 & \text{if } z_i < \bar{z} \\ 1 & \text{if } z_i > \bar{z} \end{cases}$$

and distributing the elements $z_i = \bar{z}$ in such a way that the balance is preserved.

T. Chan, P. Ciarlet and W. Szeto showed in [9] that the vector x formed that way is closest in any α-norm to z amongst all balanced ±1-vectors, i.e.,

$$x = \operatorname*{arg\,min}_{p \in \{\pm 1\}^n,\ p^T e = 0} \|p - z\|_\alpha$$

Note that in both cases, it is not necessary to compute the eigenvector to a high degree of accuracy. If we use an iterative method (like the Lanczos algorithm [25, Ch. 9] or the Rayleigh quotient iteration (RQI) ([25, Sec. 8.2.3])), we can stop iterating once the result is sufficiently accurate to either decide on the sign of the elements or to recognize the $n/2$ largest elements. This can be much cheaper than computing the exact eigenvector.

3.3.1 Spectral Bisection for weighted graphs

It is easy to generalize the spectral bisection algorithm to handle weighted graphs.

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with weighted edges $W_\mathcal{E}$, one can define the *weighted Laplacian* $L = (l_{ij})_{ij}$ with

$$l_{ij} = \begin{cases} -w_e(e_{i,j}) & \text{if } i \neq j \text{ and } e_{i,j} \in \mathcal{E} \\ 0 & \text{if } i \neq j \text{ and } e_{i,j} \notin \mathcal{E} \\ \sum_{k=1}^{n} w_e(e_{i,k}) & \text{if } i = j \end{cases}$$

With this definition and x as above (see equation (3.2)), $\frac{1}{4}x^T Lx$ still denotes the weight of the cut edges and so we can proceed as above.

If the graph has weighted vertices $W_\mathcal{V}$, the subsets have equal weights if $w^T x = 0$ with $w = (w_v(v_i))_{i=1}^n$ the vector of the vertex weights. Using $V = \operatorname{diag}(w)$ (a diagonal matrix with the components of w on the diagonal), we reformulate this condition as $x^T V e = 0$. We therefore have to solve the problem

$$\begin{array}{ll} \text{Minimize} & f(x) = \frac{1}{4}x^T Lx \\ \text{subject to} & x \in \{\pm 1\}^n \\ & x^T V e = 0 \end{array} \tag{3.7}$$

Again we relax the constraints a bit: instead of choosing discrete values for the x_i we now look for a real vector z which has the same (weighted) length as x, i.e., $z^T V z = e^T V e (= \sum_{i=1}^n w_v(v_i))$ (note that the feasible solutions of the discrete problem (3.7) satisfy this condition) and fulfills the other conditions:

$$\begin{array}{ll} \text{Minimize} & f(z) = \frac{1}{4}z^T Lz \\ \text{subject to} & z^T V z = e^T V e \\ & z^T V e = 0 \end{array} \tag{3.8}$$

A solution for the relaxed equation (3.8) is the eigenvector corresponding to the second smallest eigenvalue of the generalized eigenproblem $Lx = \lambda V z$. But since V is diagonal with positive elements on the diagonal, the problem can be reduced to an ordinary eigenproblem by substituting $y = V^{1/2}z$. This leads to $Ay = \lambda y$ with $A = V^{-1/2} L V^{-1/2}$. The new matrix A is also positive semidefinite with the same structure as L.

3.3.2 Spectral Quadri– and Octasection

As seen in section 2.2, it can be advantageous to divide a graph into more than two parts at once, so it would be interesting to find a spectral algorithm doing this. Rendl and Wolkowitz describe one such technique in [55]. While their algorithm allows the decomposition of a graph in p parts, it requires the calculation of p eigenvectors and is thus expensive. But as we will see, the spectral bisection idea described above can be also adapted to divide a graph into more than two parts at once. Hendrickson and Leland [30, 34] developed a way to partition a graph into four (or eight) partitions of equal size at once using two (or three) eigenvectors, i.e., spectral *quadrisection* (resp. *octasection*). R. Van Driessche [61] later generalized the quadrisection algorithm to divide the graph into partitions of unequal size.

Contrary to spectral bisection, these algorithms do not minimize the cut-size but a closely related size, the *hypercube hop* or *Manhattan* metric. In this, the weight of each cut edge is multiplied by the number w of bits that differ in the binary representation of the subset numbers (see Figure 3.2). This models the performance of parallel communication on hypercube and 2- or 3-dimensional mesh topologies (cf. [28]). In these topologies, w can be interpreted as the number of wires a message must take while being transfered between the respective processors.

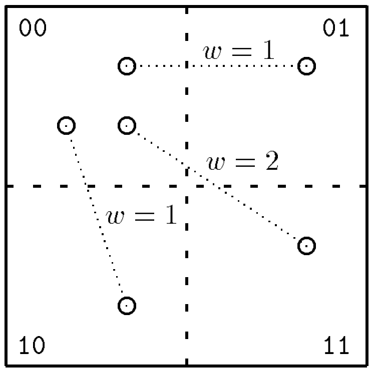


Figure 3.2: Hypercube hop metric in 2D

We will describe only the derivation of the quadrisection. For the similar but slightly more complicated octasection, see [30] or [34].

To distinguish between the four subpartitions $\mathcal{V}_0, \ldots, \mathcal{V}_3$, we will use two index vectors $x, y \in \{\pm 1\}^n$ with

$$\begin{aligned}
x_i = -1 \text{ and } y_i = -1 &\iff v_i \in \mathcal{V}_0 \\
x_i = -1 \text{ and } y_i = +1 &\iff v_i \in \mathcal{V}_1 \\
x_i = +1 \text{ and } y_i = -1 &\iff v_i \in \mathcal{V}_2 \\
x_i = +1 \text{ and } y_i = +1 &\iff v_i \in \mathcal{V}_3
\end{aligned}$$

Using considerations similar to the ones that that lead to (3.3), we find that

$$f(x, y) = \tfrac{1}{4}(x^T L x + y^T L y)$$

gives us the hop weight of the partition induced by x and y. Assuming that n is a multiple of 4, the constraints that all $\mathcal{V}_i$ are to be of the same size can be expressed as

$$\sum_i (1-x_i)(1-y_i) = (e-x)^T(e-y) = n$$
$$\sum_i (1-x_i)(1+y_i) = (e-x)^T(e+y) = n$$
$$\sum_i (1+x_i)(1-y_i) = (e+x)^T(e-y) = n$$
$$\sum_i (1+x_i)(1+y_i) = (e+x)^T(e+y) = n$$

which (after expanding and solving a small linear system) simplifies to

$$e^T x = 0 \qquad e^T y = 0 \qquad x^T y = 0.$$

Therefore we want to solve the following discrete minimization problem

$$\begin{aligned} \text{Minimize} \quad & f(x,y) = \tfrac{1}{4}(x^T L x + y^T L y) & \\ \text{subject to} \quad & e^T x = e^T y = x^T y = 0 & (3.9) \\ & x, y \in \{\pm 1\}^n & (3.9.a) \end{aligned}$$

Again, since this problem is too difficult to solve, we relax the discreteness condition (3.9.a) and only demand that the solutions $z, w \in \mathbb{R}^n$ have the same Euclidean length ($\sqrt{n}$) as x, y. We then get

$$\begin{aligned} \text{Minimize} \quad & f(z,w) = \tfrac{1}{4}(z^T L z + w^T L w) & \\ \text{subject to} \quad & e^T z = e^T w = z^T w = 0 & (3.10) \\ & z^T z = w^T w = n & (3.10.a) \end{aligned}$$

Using an argument similar to the one in the bisection case (p. 32 ff.), we see that the scaled eigenvectors $z = \sqrt{n} u_2$ and $w = \sqrt{n} u_3$ belonging to the second- and third-smallest eigenvalues λ_2 and λ_3 are a solution of the continuous problem (3.10) and $\frac{n}{4}(\lambda_2 + \lambda_3)$ is a lower bound on the number of hops induced by any quadrisection of $\mathcal{G}$.

But any orthonormal pair $\bar{z}, \bar{w}$ from the subspace spanned by λ_2 and λ_3 (i.e., $\bar{z} = \sqrt{n} u_2 \cos\theta + \sqrt{n} u_3 \sin\theta$ and $\bar{w} = -\sqrt{n} u_2 \sin\theta + \sqrt{n} u_3 \cos\theta$) are also solutions of (3.10). The question now remains which of the solutions to choose. One idea is to choose θ in such a way that the resulting values closest to the discrete values ± 1 that we were originally looking for, i.e., choose a θ_0 that minimizes $g(\theta) = \sum_i (1-\bar{z}_i^2)^2 + (1-\bar{w}_i^2)^2$ for $\theta \in [0, 2\pi)$.

$g(\theta)$ reduces to a quartic expression in $\sin(\theta)$ and $\cos(\theta)$. After constructing $g(\theta)$ (at a cost of $\mathcal{O}(n)$), the cost of evaluating (and thus minimizing) $g(\theta)$ is independent of n and usually insignificant compared to calculating the eigenvectors. In [30] it is recommended to use a local minimizer with some random starting points.

But of course originally we were looking for an approximative solution to the discrete problem (3.9). So the last hurdle is the assignment of the continuous points (z_i, w_i) to discrete points $(\pm 1, \pm 1)$. Since we need to adhere to the restrictions given in (3.9), i.e., the partitions need to have the same size, we cannot simply use the median method we used for the bisection case. If we define a distance function from continuous points (z_i, w_i) to discrete points $(\pm 1, \pm 1)$, distributing the points evenly while minimizing the sum of the distance is known as the *minimum cost assignment* problem. According to [29] there exist efficient algorithms to solve the problem.

3.3.3 Multilevel Spectral Bisection

For the above algorithms, we have to compute the Fiedler-vector or the eigenvector associated with the third-smallest eigenvalue (and, for octasection, the forth-smallest) of a sparse, symmetric matrix. First choice for such an algorithm is some variation of the Lanczos-algorithm ([25, Ch. 9]). But it turns out that compared to other graph partitioning algorithms, spectral bisection using the Lanczos-algorithm is extremely slow.

But how to speed up the calculation of the eigenvector(s) and thus the spectral bisection algorithm? Luckily, we do not need to calculate the eigenvectors of just any sparse matrix but only those of the Laplacian of a graph. In [3], S. Barnard and H. Simon found a clever way to profit from the connections between the eigenvectors and the graph. We will present their idea in this section.

The basic idea is quite simple. Given a graph $\mathcal{G}^{(0)}$, we try to contract it to a new graph $\mathcal{G}^{(1)}$ that is in some sense similar to $\mathcal{G}^{(0)}$ but with fewer vertices. Since the new graph (and consequently its Laplacian) is smaller, it is cheaper to calculate the Fiedler-vector $u_2^{(1)}$ of the new graph. We then use $u_2^{(1)}$ as a starting point to calculate the Fiedler-vector $u_2^{(0)}$ of the original graph. This process has also been termed *coarsening the graph* and the smaller graph is also called *coarser.*

Of course, we can use this idea recursively, constructing ever smaller graphs $\mathcal{G}^{(2)}, \mathcal{G}^{(3)}, \ldots$ to calculate $u_2^{(1)}$, $u_2^{(2)}$ and so on. These different sized graphs lead to the name of this algorithm: *Multilevel Spectral Bisection* or *MSB.*

So the first, rough draft of the eigenvector calculation for MSB looks like this (again, we present it only for an unweighted graph; generalization to the case with weighted edges is quite straightforward):

ALGORITHM 3.5 (MULTILEVEL FIEDLER-VECTOR CALCULATION)

Function *Fiedler*($\mathcal{G}$)

If $\mathcal{G}$ is small enough **then**

Calculate $f = \mathsf{u}_2$ using the Lanczos algorithm

else

Construct the smaller graph $\mathcal{G}'$ and (implicitly) the corresponding Laplacian L'.

f′ = *Fiedler*($\mathcal{G}'$)

Use f′ to find an initial guess $f^{(0)}$ for f.

calculate f from the initial guess $f^{(0)}$.

endif

Return f

Of course, we have left out three rather important details: How to construct the smaller graph, how to find an initial guess $f^{(0)}$ for f and how to efficiently calculate f using $f^{(0)}$.

Constructing the smaller graph

First, we need the following definitions:

DEFINITION 3.3 Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a subset $\mathcal{V}' \subset \mathcal{V}$ is called an *independent set* with respect to $\mathcal{V}$ if no two elements of $\mathcal{V}'$ are connected by an edge.

DEFINITION 3.4 $\mathcal{V}'$ is called a *maximal independent set* with respect to $\mathcal{V}$ if adding any vertex from $\mathcal{V} \setminus \mathcal{V}'$ to $\mathcal{V}'$ would no longer make it an independent set.

(Note that there is another definition of a maximal independent set that requires the independant set to be of maximal size, i.e., $\mathcal{V}'$ is a maximal independent set with respect to $\mathcal{V}$ if there does not exist an independant subset $\mathcal{V}''$ of $\mathcal{V}$ with $|\mathcal{V}'| < |\mathcal{V}''|$. The two definitions are *not* equivalent.)

It is easy to see that an independent set $\mathcal{V}' \subset \mathcal{V}$ is maximal if and only if every node from $\mathcal{V} \setminus \mathcal{V}'$ is adjacent to a node in $\mathcal{V}$.

A maximal independent set can easily and efficiently be found by using a greedy algorithm and while they are by no means unique, different maximal independent sets typically have a similar size.

So, to coarsen a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ to a new, smaller graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$, we do the following:

- Choose a maximal independent subset $\mathcal{V}'$ of $\mathcal{V}$ and use it as the vertex set of the new graph. (any subset of $\mathcal{V}$ would do, but a maximally independent set is guaranteed to be evenly distributed over $\mathcal{G}$.)
- Grow domains in $\mathcal{G}$ around the vertices in $\mathcal{V}'$ and add an edge e'_{ij} to $\mathcal{E}'$ whenever the domains around v'_i and v'_j intersect

Growing a domain around a vertex is done by repeatedly adding all neighbors of vertices already in the domain. If the starting set is maximal independent, two cycles of domain growth are enough to cover the whole graph and therefore determine $\mathcal{E}'$. In that case, two vertices in $\mathcal{V}'$ are connected exactly if there exists a path with 4 vertices or less connecting the corresponding vertices in $\mathcal{V}$. It is also easy to see that the resulting graph is connected if and only if the original one was connected as well [4] (Note that contrary to what is stated in [4], two growth cycles are enough since (using the notation in [4]) $G^{[k]}$ "shortcuts" all paths of length 2^k instead of the claimed $k+1$).

Finding an initial guess

Having calculated the Fiedler vector f' of the smaller graph, we now face the problem of how to use it to approximate the Fiedler vector f of the bigger graph. Here we use our knowledge about the relationship between $\mathcal{G}$ and $\mathcal{G}'$. First we assign the values f_i to the appropriate elements of f. In mathematical terms: Let $m(i)$ be a mapping between the vertices of the condensed graph and the vertices of $\mathcal{G}$ such that i is the index of the vertex in $\mathcal{V}'$ that corresponds to the $m(i)$th vertex in $\mathcal{V}$. Then let

$$f_{m(i)} = f'_i \text{ for } i = 1, \ldots, |\mathcal{V}'|$$

Finally, for the components j of f corresponding to vertices in $\mathcal{V} \setminus \mathcal{V}'$, we assign the average of the values corresponding to those vertices in $\mathcal{V}'$ adjacent to v_j. The mathematical formulation of this is:

$$f_j = \sum_{\substack{i:\, v_i \in \mathcal{G}' \\ e_{i,j} \in \mathcal{E}}} f_i \Big/ \sum_{\substack{i:\, v_i \in \mathcal{G}' \\ e_{i,j} \in \mathcal{E}}} 1 \quad \text{for all } j \text{ with } v_j \in \mathcal{V} \setminus \mathcal{V}'$$

Since $\mathcal{V}'$ is maximal independent, every element in $\mathcal{V} \setminus \mathcal{V}'$ has at least one neighbor in $\mathcal{V}'$. With the above we have thus assigned a value to each component of $f^{(0)}$.

One should note that this operation is similar to the *prolongation* operation in classical multigrid methods for linear systems [7].

Calculating the Fiedler-vector using the estimate $\mathbf{f^{(0)}}$

The last detail to solve is: how to calculate the Fiedler-vector f using the estimate $f^{(0)}$. While the Lanczos-algorithm is the usual favorite for extreme eigenpairs of symmetric sparse matrices, it does not effectively take advantage of an initial guess. In [3], the Rayleigh quotient iteration (RQI) ([25, Sec. 8.2.3]) was choosen to refine the initial guess $f^{(0)}$. Since we already have a good approximation and our accuracy demands are rather low, RQI will usually take only very few iterations to converge.

Algorithm 3.6 (Rayleigh quotient iteration)
Given an initial guess $(f^{(0)}, \lambda^{(0)})$ for an eigenpair (f, λ), set $v = f^{(0)}/\|f^{(0)}\|$ and

repeat
$\phi := v^T L v$
Solve $(L - \phi I)x = v$
$v := x/\|x\|$
until ϕ is "close enough" to λ

A closer look at the RQI algorithm reveals that we have to solve a linear equation with $L-\phi I$ in each step. Since this matrix is sparse, symmetric but probably indefinite and ill-conditioned, the method of choice is the SYMMLQ iteration [48]. According to [3] the total number of matrix-vector operations in all RQI steps combined is generally less than 10.

In experiments (described in [3]), the MSB with RQI and SYMMLQ has turned out to be very efficient. Compared with the normal spectral bisection using the Lanczos algorithm, it usually is at least an order of magnitude faster.

3.3.4 Algebraic Multilevel

In the MSB algorithm described above, the smaller graph was constructed solely out of the larger graph and only then the corresponding Laplacian matrix was (implicitly) build and its Fiedler-vector computed.

But the problem can also be approached in another way, more in tune with the classical multilevel algorithms for linear systems [7]. That is: given a Laplacian matrix $L \in \mathbb{R}^{x \times n}$, we construct a restriction operator $R \in \mathbb{R}^{n \times n'}$ and a prolongation operator $P \in \mathbb{R}^{n' \times n}$ and study the relation between (prolongated) solution of the smaller, restricted problem and the original problem. In [61], R. Van Driessche did just that. We will report some of his results in this section.

Also choosing a maximal independent subset of $\mathcal{V}$, the matrix equivalent of the interpolation method above (in section 3.3.3) was used as prolongation operator P.

Further two restriction operators were choosen, $R^P = P^T$ and R^I a simple injection operator ($r^I_{ij} = \delta_{m(i),j}$) for $i = 1, \dots, n'$ and $j = 1, \dots, n$.

It turns out that using R^I leads to another eigenproblem with a matrix $L' = R^I LP$. L can be interpreted as the Laplacian matrix of a new, smaller graph with weighted edges. This new graph can be shown to be connected, so we can repeat the procedure recursively.

R^P does not have these nice properties. Using it leads to a generalized eigenproblem. $L' = R^P LP$ is again the Laplacian of a weighted graph but that graph cannot be guaranteed to be connected and the weights of the edges might not be positive. A repeated application may therefore not be possible. Finally, the solution of the generalized eigenproblem is computationally more expensive.

In [61], the prolongations v^I_i and v^P_i of the second, third and fourth eigenvector of the contracted graph were compared with the corresponding eigenvectors $u_j, (j = 2, 3, 4)$ of the original graph. This was done by examining the absolute values of the inner products of the (normalized) vectors. Notable results were:

- The prolongations v^I_i, v^P_i were quite rich in u_i $(i = 2, 3, 4)$, that is the absolute value of the inner product was quite big (well above 0.95 for $j = 2, 3$ and above 0.8 for $j = 4$).

- The contributions of the other eigenvectors (i.e., for $i \neq j$) were much smaller (most of the time les than 0.1).

This means that the prolonged eigenvectors are good approximations of the eigenvector of the original graph and are thus good starting vectors for the Rayleigh quotient iteration.

Further findings included:

- The method using R^P delivered better eigenvector approximations. But this does not offset the more expensive solution of the generalized eigenproblem and so the other method (using R^I) may often be faster in delivering the final result.

- The quality of the eigenvector approximation is not only determined by the eigenvalue approximation but also by the gap between its eigenvalue and neighboring eigenvalues.

Other implementations of MSB use even other methods of constructing the smaller meshes (Chaco ([33]) for example uses one of the algorithms described in section 3.4) but report similar final results so it seems that the choice of the coarsening method is not critical.

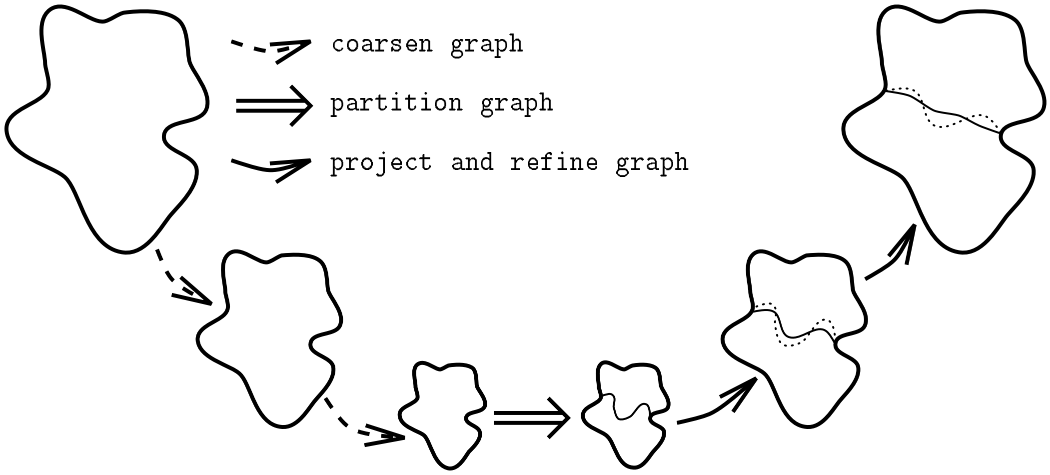


Figure 3.3: Multilevel graph partitioning

3.4 Multilevel Partitioning

The multilevel spectral bisection algorithm described in section 3.3.3 has turned out to be very efficient. Since the multilevel approach obviously has a lot of merit, can it be applied in some other way as well?

In MSB, we transfer approximate information about the second eigenvector of the original graph between levels, improve this information on each level and finally, use this eigenvector to partition the graph. A more direct way would be to simply transfer information about the partitions itself between levels and improve the partitions on each level. And this is, in a rough outline, the description of one of the most successful algorithms, the *multilevel graph partitioning*.

Algorithm 3.7 (Multilevel graph bisection)

Function *ML-Partition*($\mathcal{G}$)

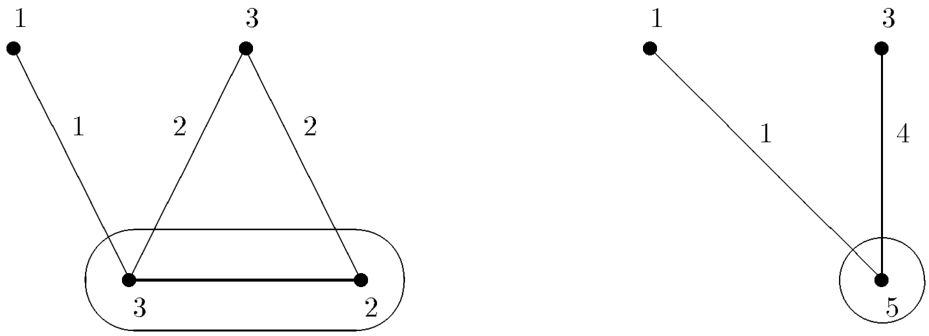
If $\mathcal{G}$ is small enough **then**

Find partition $(\mathcal{V}_1, \mathcal{V}_2)$ of $\mathcal{G}$ in some way.

else

Construct a smaller approximation $\mathcal{G}'$
$(\mathcal{V}_1', \mathcal{V}_2')$ = *ML-Partition*($\mathcal{G}'$)
$(\tilde{\mathcal{V}}_1, \tilde{\mathcal{V}}_2)$ = Project-partition-up $(\mathcal{V}_1', \mathcal{V}_2')$
$(\mathcal{V}_1, \mathcal{V}_2)$ = Refine-partition $(\tilde{\mathcal{V}}_1, \tilde{\mathcal{V}}_2)$

endif



a) original graph w. weights b) contracted graph

Figure 3.4: Coarsening a graph

Return $(\mathcal{V}_1, \mathcal{V}_2)$

In algorithm 3.7, we bisect the graph but the same idea can be used to partition the graph into p parts at once. Actually there is a whole family of slightly different multilevel graph partitioning algorithms which differ in the three generic parts "construct smaller approximation", "project partition up" and "refie projected partition" and in the way the smallest graph is partitioned. We will describe some of the solutions implemented in software packages like Chaco ([33]), METIS ([40]) or WGPP ([26]).

All of the following coarsening algorithms lead to graphs with edge and vertex weights so we might as well start with a weighted graph $(\mathcal{V}, \mathcal{E})$.

3.4.1 Coarsening the graph

In all of the following methods, the coarser graph is constructed in similar ways. First, we find a *maximal matching*. A *matching* is a subset $\mathcal{M} \subset \mathcal{E}$ of edges no two of which share an endpoint. A matching is *maximal* if no other edges from $\mathcal{E}$ can be added.

Given the matching $\mathcal{M}$, the new graph is constructed in the following way: for each edge $e_{i,j} \in \mathcal{M}$, we contract its endpoints v_i and v_j into a new vertex v_n. Its weight equals the sum $w_v(v_i) + w_v(v_j)$ of the weights of v_i and v_j. Its neighbors are the combined neighbors of v_i and v_j with accumulated edge-weights, i.e., $w_e(e_{n,k}) = w_e(e_{i,k}) + w_e(e_{i,k})$ (where the weight on nonexistent edges is assumed to be 0). The edge $e_{i,j}$ disappears. (See also Figure 3.4).

The main difference between the different implementations is the construction of the maximal matching. The simplest method (suggested by B. Hendrickson and R. Leland in [31]) is *randomized matching* (RM) which randomly selects fitting edges until the matching is maximal:

Algorithm 3.8 (Randomized Matching (RM))

Unmark all vertices and set $\mathcal{M} = \emptyset$.

While there are unmarked vertices left

  Randomly select unmarked vertex v_i

  Mark v_i

  If v_i has unmarked neighbors

    Randomly choose unmarked neighbor v_j of v_i $(\star)$
    Mark v_j
    Add edge $e_{i,j}$ to $\mathcal{M}$

If we look at the way the coarser graph is constructed, we notice that the edges in the matching "disappear" from the lower levels. The total edge-weight of the coarser graph equals the edge-weight of the original graph minus the weight of the edges in the matching. In [39] it is suggested that by decreasing the total edge-weight of the coarser graph one might lower its cutsize and also the cutsize of the original graph. Experiments confirmed this assumption and also showed that this choice accelerated the refinement process.

To achieve this, G. Karypis and V. Kumar use a slight modification of the RM algorithm called *heavy edge matching* (HEM). Instead of choosing a random neighbor (in line $(\star)$ of Algorithm 3.8), they simply select the unmarked neighbor which is connected to v_i by the heaviest edge.

The problem with this approach is that it might miss some heavy edges. If in Figure 3.5 vertex v_i is selected first, the edge with weight 3 will be selected. Consequently the heavier edge with weight 7 will not be selected. To avoid this, A. Gupta suggests in [27] to sort the edges by weight and then to choose the heaviest permissible edge first (ties are broken randomly). He calls this method is *heaviest edge matching.* Is usually done only in the later stages of the coarsening process when the graphs are considerably smaller and thus the sorting is cheap. Furthermore, heaviest edge matching is particularly efficient when the graph has edges with very different weights. This weight inequilibrium will often appear after some coarsening steps.

These basic strategies can be coupled with other methods to further improve the coarser graph.

G. Karypis and V. Kumar try to decrease the average degree of the coarser graph, arguing that fewer edges lead to a higher average edge-weight which in turn will help the HEM to find heavier edges. This *modified heavy edge matching* (HEM*) is done by choosing the appropriate edge amongst all permissible edges with maximal weight.

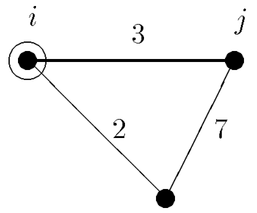


Figure 3.5: Missing a heavy edge

Using his *heavy-triangle matching* (HTM), A. Gupta also tries to contract three vertices into one, again trying to maximize the weight of the connecting edges. Additionally, he tries to keep the weights of the coarsened vertices balanced by using a modified edge-weight $\tilde{w}_e(e_{i,j}) = \alpha w_e(e_{i,j}) + \beta(w_v(v_i) + w_v(v_j))^{-1}$ which also takes the vertex-weights into account. For this, the graph is regarded as completely connected with the additional edges having zero weight. This simplifies the bisection of the final graph.

3.4.2 Partitioning the smallest graph

Once the graph is coarse enough (usually when $|\mathcal{V}'|$ is less than say 50 or 100 or when further coarsening steps only reduce the graph a little), it needs to be partitioned. To do this, one can use any of the other methods mentioned above. Since the graph to be partitioned is very small, efficiency is not important. One can even try several methods (or the same randomized method multiple times) and choose the best result. Methods normally used include graph-growing, spectral bisection or just Kernigan–Lin with random starting partitions.

But this very small graph will have vertices with (sometimes very) different weights. So it might be impossible to divide the vertices into p partitions of equal weight. And even if it is possible, doing so might cause a high cutsize. So it is advisable to allow some imbalance in order to improve the quality of the cut. This imbalance can then be reduced during the refinement steps.

The WGPP program ([26]) uses a clever way to eliminate the need for additional code to partition the smallest graph. Since its coarsening method tries to keep the weights of the coarsened vertices balanced, it simply coarsens the graph down to p (normally 2) vertices and then uncoarsens and refines it up to q (with $p < q \ll n$). It then saves this partition, coarsens it again down to p vertices (remember that this coarsening is randomized) and repeats this process, always saving the currently best partition. The quality of the partition should also take the partition size imbalance into consideration. Since q is small, the process can be cheaply repeated a couple of times and the best result is chosen.

3.4.3 Projecting up and refining the partition

Given the partition of the coarser graph, we now need to find a good partition of the finer graph. First, we "project up" the partition of the coarser graph to the finer one. Each vertex of the finer graph can be traced to exactly one vertex in the coarser graph: either it (together with one (or more) other vertices) is contracted into a new coarser-level vertex or it is simply copied to the coarser level. So having partitioned the coarser graph, we can simply assign the vertices on the finer graph to the appropriate partitions.

But on the finer graph we now have more "degrees of freedom" to find a better partition (with a lower cutsize), so we need to refine the graph, improving both the cutsize and the balance (since the partitions might be of different weight).

The refinement is usually done by using one of the variants of the Kernigan–Lin algorithm described in section 3.2.1. The Fiduccia–Mattheyses variant [18] in particular can easily be modified to improve the balance at the same time by adding some term describing the imbalance into the cost-function used to find the next vertex to move. Other changes include using only one inner pass and considering only "boundary-vertices".

3.5 Other methods

In the sections above, we have described the more common methods for graph partitioning. But there are many other methods as well, both geometrically oriented and graph based, which we will only cite here ([8, 14, 16, 55, 63, 65–67]). Some of the methods are specialized for finite element meshes for PDEs ([13, 60, 62]). Others have tried to apply general optimization techniques like simulated annealing or genetic algorithms to the partitioning problem ([36, 37, 61, 68]). And finally, in addition to the software libraries mentioned above, there are other programs available as well, for example Jostle ([64]), Party ([54]) and Scotch ([49]).

Bibliography

[1] G. Ahmdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Proc. of the SJCC*, volume 30, pages 483–485, 1967.

[2] S. T. Barnard, A. Pothen, and H. Simon. A spectral algorithm for envelope reduction of sparse matrices. *Numerical Linear Algebra with Applications*, 2(4):317–334, 1995.

[3] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6:101–107, Feb. 1994.

[4] S. T. Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*. ACM / IEEE, 1995. published online and as CD.

[5] A. Berman and R. J. Plemmons. *Nonnegative Matrices in the Mathematical Sciences*. Academic Press, 1979.

[6] M. W. Berry, B. Hendrickson, and P. Raghavan. Sparse matrix reordering schemes for browsing hypertext. In J. Renegar, M. Shub, and S. Smale, editors, *The Mathematics of Numerical Analysis*, volume 32 of *Lectures in Applied Mathematics*, pages 99–123. AMS, 1996.

[7] W. L. Briggs. *A Multigrid Tutorial*. SIAM, 1987.

[8] T. F. Chan, J. R. Gilbert, and S.-H. Teng. Geometric spectral partitioning. Technical Report CSL-94-15, Parc Xerox, Jan. 1995.

[9] T. F. Chan, J. P. Ciarlet, and W. K. Szeto. On the optimality of the median cut spectral bisection graph partitioning method. *SIAM Journal on Scientific Computing*, 18(3):943–948, May 1997.

[10] T. F. Chan and W. K. Szeto. A sign cut version of the recursive spectral bisection graph partitioning algorithm. In J. G. Lewis, editor, *Proceedings of the 5th SIAM Conference on Applied Linear Algebra (Snowbird UT)*. SIAM, 1994.

[11] G. Chartrand and O. R. Oellermann. *Applied and algorithmic graph theory*. International series in pure and applied mathematics. McGraw-Hill, New York, 1993.

[12] P. G. Ciarlet and J. L. Lions. *Handbook of numerical analysis; Part 2: Finite element methods*. North-Holland, Amsterdam, 1992.

[13] H. L. deCougny, D. D. Devine, J. E. Flaherty, R. M. Loy, C. Özturan, and M. S. Shephard. Load balancing for the parallel adaptive solution of partial differential equations. *Applied Numerical Mathematics*, 16:157–182, 1994.

[14] R. Dieckmann, B. Monien, and R. Preis. Using helpful sets to improve graph bisections. Technical Report tr-rf-94-008, Fak. f. Informatik, Universität-GH Paderborn, June 1994.

[15] R. Dieckmann, B. Monien, and R. Preis. Load balancing strategies for distributed memory machines. Technical Report tr-rsfb-97-050, SFB 376, Universität-GH Paderborn, Sept. 1997.

[16] J. Falkner, F. Rendl, and H. Wolkowitz. A computational study of graph partitioning. *Mathematical Programming*, 66(2):211–239, 1994. Also University of Waterloo Tech.Report CORR-92-25, 1993.

[17] C. Farhat. A simple and efficient automatic FEM domain decomposer. *Computers and Structures*, 28(5):579–602, 1988.

[18] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. Technical Report 82CRD130, General Electric Co, Corporate Research and Development Center, Schenectady,NY, May 1982.

[19] M. Fiedler. Algebraic connectivity of graphs. *Czecheslovak Mathematical Journal*, 23(98):298–305, 1973.

[20] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czecheslovak Mathematical Journal*, 25(100):619–633, 1975.

[21] G. C. Fox, R. D. Williams, and P. C. Messing. *Parallel Computing Works*. Morgam Kaufmann Publishers, 1994.

[22] R. Freund and N. Nachtigal. Qmr: A quasi-minimal residual method for non-hermitian linear systems. *Numerical Mathematics*, 60:315–339, 1991.

[23] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–Completeness*. Freeman, 1979.

[24] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. Technical Report CSL-94-13, Xerox PARC, 1994. to appear: SIAM J Sci. Comp.

[25] G. H. Golub and C. F. v. Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore and London, third edition, 1996.

[26] A. Gupta. *WGPP: Watson Graph Partitioning (and sparse matrix ordering) Package.* IBM Research Division, T.J. Watson Research Center, P.O.Box 218, Yorktown Heights, New York 10598, 1996.

[27] A. Gupta. Graph partitioning based sparse matrix orderings for interior point algorithms. *IBM Journal of Research and Development*, 41(1/2):171–184, 1997.

[28] S. Hammond. *Mapping unstructured grid computations to massively parallel computers.* PhD thesis, Rensselaer Polytechnic Institute, Dept. of Computer Science, Renssselaer, NY, 1992.

[29] B. Hendrickson and R. Leland. Multidimensional spectral load balancing. In *Proc. 6th SIAM Conf. Par. Proc. Sci. Comput.*, 1993.

[30] B. Hendrickson and R. Leland. Multidimensional spectral load balancing. Technical Report SAND93-0074, Sandia National Laboratories, 1993. also in Proc. 6th SIAM Conf. Par. Proc. Sci. Comput.

[31] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993. appeared in Proc. Supercomputing 95.

[32] B. Hendrickson and R. Leland. An empirical study of static load balancing algorithms. In *Proceedings of the Scalable High Performance Computer Conference*, pages 682–685. IEEE, 1994.

[33] B. Hendrickson and R. Leland. *The Chaco User's Guide Version 2.* Sandia National Laboratories, Albuquerque NM, 1995.

[34] 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.

[35] R. A. Horn and C. R. Johnson. *Matrix Analysis.* Cambridge University Press, 1985.

[36] M. Jerrum and G. Sorkin. Simulated annealing for graph bisection. Technical Report ECS-LFCS-93-260, Dept. of Computer Science, University of Edinburgh, 1993.

[37] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part 1, graph partitioning. *Operations Research*, 37(6):865–892, Nov. 1989.

[38] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. Technical report, University of Minnesota, Department of Computer Science, 1997. short version in 34th Design Automation Conference.

[39] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, University of Minnesota, Department of Computer Science, July 1995. to appear SIAM J. Sci. Comp.

[40] G. Karypis and V. Kumar. *MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, Aug. 1995.

[41] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report TR 95-036, University of Minnesota, Department of Computer Science, 1996.

[42] B. Kernigan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 29(2):291–307, 1970.

[43] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin/Cummings, Redwood City, CA, 1994.

[44] L.-T. Liu, M.-T. Kuo, S.-C. Huang, and C.-K. Cheng. A gradient method on the initial partition of Fiduccia–Mattheyses algorithm. In *IEEE/ACM Int. Conf. on Computer-Aided Design*, pages 229–234. IEEE/ACM, Nov. 1995.

[45] R. Merris. Laplacians matrices of graphs: A survey. *Linear Algebra and its Applications*, 197, 198:143–176, 1994.

[46] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Automatic mesh partitioning. In A. George, J. R. Gilbert, and J. W. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 of *IMA Volumes in Mathematics and its Applications*, pages 57–84. Springer, 1993.

[47] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Geometric separators for finite element meshes. Technical report, University of Minnesota, 1994. to appear SIAM J. Sci. Comp, Vol.19, Nr.2, march 1998.

[48] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 12:617–629, 1975.

[49] F. Pellegrini. *SCOTCH 3.1 Users Guide.* Université Bordeaux I, Aug. 1996.

[50] A. Pothen, E. Rothberg, H. Simon, and L. Wang. Parallel sparse Cholesky factorization with spectral nested bisection ordering. Technical Report RNR-94-011, Numerical Aerospace Simulation Facility, NASA Ames Research Center, May 1994.

[51] A. Pothen, H. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(3):430–452, July 1990.

[52] A. Pothen, H. Simon, and L. Wang. Spectral nested disection. Technical Report RNR-92-003, NASA Ames Research Center, 1992. Short version in Supercomputing '92, pp. 42–51, Nov. 1992.

[53] A. Pothen. Graph partitioning algorithms with applications to scientific computing. In D. E. Keyes, A. H. Sameh, and V. Venkatakrishnan, editors, *Parallel Numerical Algorithms*. Kluwer Academic Press, 1995.

[54] R. Preis and R. Dieckmann. *The PARTY Partitioning – Library User Guide – Version 1.1*. SFB 376, Univ. Paderborn.

[55] F. Rendl and H. Wolkowicz. A projection technique for partitioning the nodes of a graph. *Annals of Operations Research*, 58:155–179, 1995.

[56] E. Rothberg and B. Hendrickson. Sparse matrix ordering methods for interior point linear programming. Technical Report SAND96-0475J, Sandia National Laboratories, Jan. 1996.

[57] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[58] Y. Saad. *Iterative methods for sparse linear system*. PWS Publishing Company, Boston, MA, 1996.

[59] H. Simon and S.-H. Teng. How good is recursive bisection. *SIAM Journal on Scientific Computing*, 18(5):1436–1445, Sept. 1997.

[60] D. Vanderstraeten and R. Keunings. Optimized partitioning of unstructured finite element meshes. *International Journal for Numerical Methods in Engineering*, 38:433–450, 1995.

[61] R. Van Driessche. *Algorithms for static and dynamic Load Balancing on Parallel Computers*. PhD thesis, KU Leuven, Nov. 1995.

[62] C. Walshaw and M. Berzins. Dynamic load balancing for PDE solvers on adaptive unstructured meshes. Research Report Series 92.32, University of Leeds School of Computer Studies, Dec. 1992.

[63] C. Walshaw, M.Cross, M. G. Everett, S.Johnson, and K. McManus. Partitioning & mapping of unstructured meshes to parallel machine topologies. In A. Ferreira and J. Rolim, editors, *Irregular 95: Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *LNCS*, pages 121–126. Springer, 1995.

[64] C. Walshaw, M.Cross, M. G. Everett, and S.Johnson. JOSTLE: Partitioning of unstructured meshes for massively parallel machines. In N. S. et al, editor, *Parallel Computational Fluid Dynamics: New Algorithms and Applications*, pages 273–280, Amsterdam, 1994. Elsevier.

[65] C. Walshaw, M.Cross, and M. G. Everett. Dynamic mesh partitioning: a unified optimisation and load-balancing algorithm. Technical Report 95/IM/03, School of Mathematics, Statistics & Computer Science, University of Greenwich, 1995.

[66] C. Walshaw, M.Cross, and M. G. Everett. A localised algorithm for optimising unstructured mesh partitions. *International Journal of Supercomputer Applications*, 9(4):280–295, 1995.

[67] C. Walshaw, M.Cross, and M. G. Everett. A parallelisable algorithm for optimising unstuctured mesh partitions. Technical Report 95/IM/06, School of Mathematics, Statistics & Computer Science, University of Greenwich, Jan. 1995.

[68] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency*, 3:457–481, 1991.