METIS*
A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices
Version 5.1.0

George Karypis

Department of Computer Science & Engineering
University of Minnesota
Minneapolis, MN 55455

karypis@cs.umn.edu

March 30, 2013

Metis [MEE tis]: *'Metis' is the Greek word for wisdom. Metis was a titaness in Greek mythology. She was the consort of Zeus and the mother of Athena. She presided over all wisdom and knowledge.*

 Related papers are available via WWW at URL: **http://www.cs.umn.edu/~karypis**

Contents

1 Introduction

Algorithms that find a good partitioning of highly unstructured graphs are critical for developing efficient solutions for a wide range of problems in many application areas on both serial and parallel computers. For example, large-scale numerical simulations on parallel computers, such as those based on finite element methods, require the distribution of the finite element mesh to the processors. This distribution must be done so that the number of elements assigned to each processor is the same, and the number of adjacent elements assigned to different processors is minimized. The goal of the first condition is to balance the computations among the processors. The goal of the second condition is to minimize the communication resulting from the placement of adjacent elements to different processors. Graph partitioning can be used to successfully satisfy these conditions by first modelling the finite element mesh by a graph, and then partitioning it into equal parts.

Graph partitioning algorithms are also used to compute fill-reducing orderings of sparse matrices. These fill-reducing orderings are useful when direct methods are used to solve sparse systems of linear equations. A good ordering of a sparse matrix dramatically reduces both the amount of memory as well as the time required to solve the system of equations. Furthermore, the fill-reducing orderings produced by graph partitioning algorithms are particularly suited for parallel direct factorization as they lead to high degree of concurrency during the factorization phase.

Graph partitioning is also used for solving optimization problems arising in numerous areas such as design of very large scale integrated circuits (VLSI), storing and accessing spatial databases on disks, transportation management, and data mining.

2 What is new in version 5.0

Version 5.0 represents nearly a complete re-write of the code-base whose purpose was to streamline and unify the stand-alone programs and API, provide better support for 64 bit architectures, enhance its functionality, and reduce the memory requirements by re-factoring its internal memory management routines. As a result, both the stand-alone programs and API routines have changed, making it incompatible with the earlier versions of METIS. However, in order to minimize the code changes that the revised API will require, the new API relies heavily on reasonable default values for most of the new parameters that it introduced.

The following represents a list of some of the major functionality-related changes and enhancements that are accessible by both the command-line programs and the API routines.

- Multi-constraint partitioning can be used in conjunction with minimization of the total communication volume.
- All graph and mesh partitioning routines take as input the target sizes of the partitions, which among others, allow them to compute partitioning solutions that are well-suited for parallel architectures with heterogeneous computing capabilities.
- When multi-constraint partitioning is used, the target sizes of the partitions are specified on a per partition-constraint pair.
- The multilevel k-way partitioning algorithms can compute a partitioning solution in which each partition is contiguous.
- All partitioning and ordering routines can compute multiple different solutions and select the best as the final solution.
- The mesh partitioning and mesh-to-graph conversion routines can operate on mixed element meshes.
- The command-line programs provide full access to the entire set of capabilities provided by METIS' API.

Operation	4.x stand-alone program	5.x stand-alone program
Partition a graph	`pmetis` `kmetis`	`gpmetis`
Partition a mesh	`partnmesh` `partdmesh`	`mpmetis`
Compute a fill-reducing ordering of a sparse matrix	`oemetis` `onmetis`	`ndmetis`
Convert a mesh into a graph	`mesh2nodal` `mesh2dual`	`m2gmetis`
Graph format checker	`graphchk`	`graphchk`

Table 1: Mapping between the old 4.x and the new 5.x command-line programs.

2.1 Changes in the command-line programs

Table 1 shows how the v4.x command-line programs map to the new set of command-line programs provided by the 5.0 version of METIS. As part of these changes, none of the functionality offered by the old programs has been removed. On the contrary, the new programs have been extended to substantially increase the type of partitionings and orderings that can be computed by them. This was achieved by expanding the number of optional parameters that these programs can take, which allow users to have a complete access to all of METIS' functionality. Prior to this release, if users wanted to access some of the advance features of METIS, they had to write their own programs based on the supplied API.

2.1.1 Migration issues

In order to support the enhanced functionality offered by the new command-line programs, the format of the graph mesh files has changed (Section 4.1). In the case of the graph file, the new format is backwards compatible, so graphs written for the earlier format will work correctly when used by `gpmetis` and `ndmetis`. However, the new mesh file format is not backward compatible, and as such they should not be used with `mpmetis` and `m2gmetis`. Fortunately, they can be easily converted to the new format by a slightly modifying their header line.

2.2 Changes in the API routines

Table 2 shows how the v4.x API routines map to the new set of APIs provided by the 5.0 version of METIS. The number of distinct core functions has been reduced to seven by expanding the calling sequence of the new routines to provide the functionally offered by the old specialized routines. In most cases, the functionality provided by the new API routines is a superset of that offered by the old routines, especially in the areas related to partitioning for heterogeneous computing architectures, multi-constraint partitioning, and communication volume-based partitioning objectives.

2.2.1 Migration issues

Since the calling sequence of all the API routines and in some cases their names has changed, migrating to the new API will require code modifications. To ensure that these modifications are minimal, the new API routines allow users to provide `NULL` as the argument to many of the parameters for which there are reasonable defaults. Thus, we expect the migration to the new API will be rather straightforward, as long as the application does not want to take advantage of the newly added capabilities.

Operation	4.x routine	5.x routine
Partition a graph	METIS_PartGraphRecursive METIS_mCPartGraphRecursive METIS_WPartGraphRecursive	METIS_PartGraphRecursive
	METIS_PartGraphKway METIS_mCPartGraphKway METIS_WPartGraphKway METIS_PartGraphVKway METIS_WPartGraphVKway	METIS_PartGraphKway
Partition a mesh	METIS_PartMeshNodal METIS_PartMeshDual	METIS_PartMeshNodal METIS_PartMeshDual
Compute a fill-reducing ordering of a sparse matrix	METIS_EdgeND METIS_NodeND METIS_NodeWND	METIS_NodeND
Convert a mesh into a graph	METIS_MeshToNodal METIS_MeshToDual	METIS_MeshToNodal METIS_MeshToDual
Utility routines	METIS_EstimateMemory	Deprecated
New utility routines		METIS_SetDefaultOptions METIS_Free

Table 2: Mapping between the old 4.x and the new 5.x API.

3 Overview of METIS

METIS is a serial software package for partitioning large irregular graphs, partitioning large meshes, and computing fill-reducing orderings of sparse matrices. METIS has been developed at the Department of Computer Science & Engineering at the University of Minnesota and is freely distributed. Its source code can downloaded directly from *http://www.cs.umn.edu/~metis*, and is also included in numerous software distributions for Unix-like operating systems such as Linux and FreeBSD.

The algorithms implemented in METIS are based on the multilevel graph partitioning paradigm [4, 3, 2], which has been shown to quickly produce high-quality partitionings and fill-reducing orderings. The multilevel paradigm, illustrated in Figure 1, consists of three phases: graph coarsening, initial partitioning, and uncoarsening. In the graph coarsening phase, a series of successively smaller graphs is derived from the input graph. Each successive graph is constructed from the previous graph by collapsing together a maximal size set of adjacent pairs of vertices. This process continues until the size of the graph has been reduced to just a few hundred vertices. In the initial partitioning phase, a partitioning of the coarsest and hence, smallest, graph is computed using relatively simple approaches such as the algorithm developed by Kernighan-Lin [5]. Since the coarsest graph is usually very small, this step is very fast. Finally, in the uncoarsening phase, the partitioning of the smallest graph is *projected* to the successively larger graphs by assigning the pairs of vertices that were collapsed together to the same partition as that of their corresponding collapsed vertex. After each projection step, the partitioning is refined using various heuristic methods to iteratively move vertices between partitions as long as such moves improve the quality of the partitioning solution. The uncoarsening phase ends when the partitioning solution has been projected all the way to the original graph.

METIS uses novel approaches to successively reduce the size of the graph as well as to refine the partition during the uncoarsening phase. During coarsening, METIS employs algorithms that make it easier to find a high-quality partition at the coarsest graph. During refinement, METIS focuses primarily on the portion of the graph that is close to the partition boundary. These highly tuned algorithms allow METIS to quickly produce high-quality partitions and fill-reducing orderings for a wide variety of irregular graphs, unstructured meshes, and sparse matrices.

Operation	Stand-alone program	API routine
Partition a graph	`gpmetis`	`METIS_PartGraphRecursive` `METIS_PartGraphKway`
Partition a mesh	`mpmetis`	`METIS_PartMeshNodal` `METIS_PartMeshDual`
Compute a fill-reducing ordering of a sparse matrix	`ndmetis`	`METIS_NodeND`
Convert a mesh into a graph	`m2gmetis`	`METIS_MeshToNodal` `METIS_MeshToDual`

Table 3: An overview of METIS' command-line and library interfaces.

METIS provides a set of stand-alone command-line programs for computing partitionings and fill-reducing orderings as well as an application programming interface (API) that can be used to invoke its various algorithms from C/C++ or Fortran programs. The list of stand-alone programs and API routines of METIS is shown in Table 3. The API routines allow the user to alter the behavior of the various algorithms and provide additional routines that partition graphs into unequal-size partitions and compute partitionings that directly minimize the total communication volume.

3.1 Partitioning a graph

METIS can partition an unstructured graph into a user-specified number k of parts using either the multilevel recursive bisection [4] or the multilevel k-way partitioning [3] paradigms. Both of these methods are able to produce high quality partitions. However, METIS's multilevel k-way partitioning algorithm provides additional capabilities (e.g., minimize the resulting subdomain connectivity graph, enforce contiguous partitions, minimize alternative objectives, etc.) and as such it may be preferable than multilevel recursive bisection. METIS' stand-alone program for partitioning

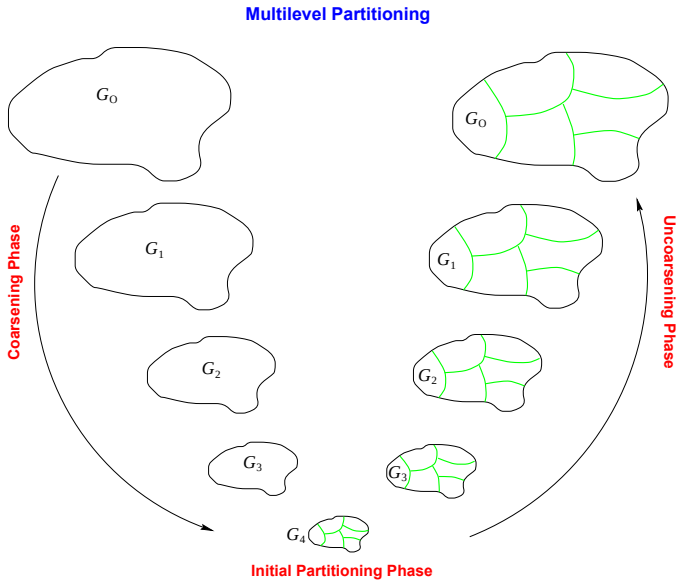


Figure 1: The three phases of multilevel k-way graph partitioning. During the coarsening phase, the size of the graph is successively decreased. During the initial partitioning phase, a k-way partitioning is computed, During the multilevel refinement (or uncoarsening) phase, the partitioning is successively refined as it is projected to the larger graphs. G_0 is the input graph, which is the finest graph. G_{i+1} is the next level coarser graph of G_i. G_4 is the coarsest graph.

a graph is `gpmetis` and the functionality that it provides is achieved by the `METIS_PartGraphRecursive` and `METIS_PartGraphKway` API routines.

3.2 Alternate partitioning objectives

The objective of the traditional graph partitioning problem is to compute a k-way partitioning such that the number of edges (or in the case of weighted graphs the sum of their weights) that straddle different partitions is minimized. This objective is commonly referred to as the *edge-cut*. When partitioning is used to distribute a graph or a mesh among the processors of a parallel computer, the objective of minimizing the edge-cut is only an approximation of the true communication cost resulting from the partitioning [1].

The communication cost resulting from a k-way partitioning generally depends on the following factors: (i) the total communication volume, (ii) the maximum amount of data that any particular processor needs to send and receive; and (iii) the number of messages a processor needs to send and receive. METIS' multilevel k-way partitioning approaches can be used to directly minimize the total communication volume resulting from the partitioning (first factor). In addition, METIS also provides support for minimizing the third factor (which essentially reduces the number of startups) and indirectly (up to a point) reduces the second factor.

3.3 Support for multi-phase and multi-physics computations

The traditional graph partitioning problem formulation is limited in the types of applications that it can effectively model because it specifies that only a single quantity be load balanced. Many important types of multi-phase and multi-physics computations require that multiple quantities be load balanced simultaneously. This is because synchronization steps exist between the different phases of the computations, and so, each phase must be individually load balanced. That is, it is not sufficient to simply sum up the relative times required for each phase and to compute a partitioning based on this sum. Doing so may lead to some processors having too much work during one phase of the computation (and so, these may still be working after other processors are idle), and not enough work during another. Instead, it is critical that every processor have an equal amount of work from each phase of the computation.

METIS includes partitioning routines that can be used to partition a graph in the presence of such multiple balancing constraints. Each vertex is now assigned a vector of m weights and the objective of the partitioning routines is to minimize the edge-cut subject to the constraints that each one of the m weights is equally distributed among the domains. For example, if the first weight corresponds to the amount of computation and the second weight corresponds to the amount of storage required for each element, then the partitioning computed by the new algorithms will balance both the computation performed in each domain as well as the amount of memory that it requires. The multi-constraint partitioning algorithms and their applications are further described in [2]. The `gpmetis` program invokes the multi-constraint partitioning routines whenever the input graph specifies more that one set of vertex weights, and all of its graph partitioning API routines allow for the specification of multiple balancing constraints.

3.4 Partitioning a mesh

METIS provides the `mpmetis` program for partitioning meshes arising in finite element or finite volume methods. This program take as input the element-node array of the mesh and compute a k-way partitioning for both its elements and its nodes. This program first converts the mesh into either a dual graph (i.e., each element becomes a graph vertex) or a nodal graph (i.e., each node becomes a graph vertex), and then uses the graph partitioning API routines to partition this graph. METIS utilizes a flexible approach for creating a graph for a finite element mesh, which allows it to handle meshes with different and possibly mixed element types (e.g., triangles, tetrahedra, hexahedra, etc.). The functionality provided by `mpmetis` is achieved by the `METIS_PartMeshNodal` and `METIS_PartMeshDual` API routines.

3.5 Partitioning for heterogeneous parallel computing architectures

Heterogeneous computing platforms containing processing nodes with different computational and memory capabilities are becoming increasingly more common. METIS' graph and mesh partitioning programs and API routines are designed to partition a graph into k parts such that each part contains a pre-specified fraction of the total number

of vertices/elements/nodes. In addition, in the case of multi-constraint partitioning, these pre-specified fractions are provided for each one of the vertex weights. By matching the weights specified for each partition to the relative computational and memory capabilities of the various processors, these routines can be used to compute partitionings that balance the computations on heterogeneous architectures.

3.6 Computing a fill-reducing ordering of a sparse matrix

METIS provides the `ndmetis` program and its associated `METIS_NodeND` API routine for computing fill-reducing orderings of sparse matrices based on the multilevel nested dissection paradigm [4]. The nested dissection paradigm is based on computing a vertex-separator for the the graph corresponding to the matrix. The nodes in the separator are moved to the end of the matrix, and a similar process is applied recursively for each one of the other two parts. The multilevel nested dissection paradigm is quite effective in producing re-orderings that incur low fill-in.

3.7 Converting a mesh into a graph

METIS provides the `m2gmetis` program for converting a mesh into the graph format used by METIS. This program can generate either the nodal or dual graph of the mesh. The corresponding API routines are `METIS_MeshToNodal` and `METIS_MeshToDual`. Since METIS does not provide API routines that can directly compute a multi-constraint partitioning of a mesh, these routines can be used to first convert the mesh into a graph, which can then be used as input to METIS' graph partitioning routines to obtain such partitionings.

4 METIS' stand-alone programs

METIS provides a variety of programs that can be used to partition graphs, partition meshes, compute fill-reducing orderings of sparse matrices, as well as programs to convert meshes into graphs appropriate for METIS's graph partitioning programs.

4.1 Input file formats

The various programs in METIS require as input either a file storing a graph or a file storing a mesh. The format of these files are described in the following sections.

4.1.1 Graph file

The primary input of the partitioning and fill-reducing ordering programs in METIS is the ***undirected*** graph to be partitioned or ordered. This graph is stored in a file and is supplied to the various programs as one of the command line parameters.

A graph $G = (V, E)$ with n vertices and m edges is stored in a plain text file that contains $n + 1$ lines (excluding comment lines). The first line, referred to as the *header line* contains information about the size and the type of the graph, while the remaining n lines contain information for each vertex of G. Any line that starts with '%' is a comment line and is skipped.

The header line contains either two (n, m), three (n, m, *fmt*), or four (n, m, *fmt*, *ncon*) parameters. The first two parameters (n, m) are the number of vertices and the number of edges, respectively. Note that in determining the number of edges m, an edge between any pair of vertices v and u is counted **only once** and not twice (i.e., we do not count the edge (v, u) separately from (u, v)). For example, the graph in Figure 2 contains 11 vertices.

The *fmt* parameter is used to specify if the graph file contains information about vertex sizes, vertex weights, and edge weights. The *fmt* parameter is a three-digit binary number. If the least significant bit is set to 1 (i.e., the 1st bit from right to left), then the graph file provides information about the weights of the edges. If the second least significant bit is set to 1 (i.e., the 2nd bit from right to left), then the graph file provides information about the weights of the vertices. Finally, if the third lest significant bit is set to 1 (i.e., the 3rd bit from right to left), then the graph file provides information of the sizes of the vertices. For example, if *fmt* is `011`, then the graph file provides information about both vertex weights and edge weights. Note that when the *fmt* parameter is not provided, it is assumed that the vertex sizes, vertex weights, and edge weights are all equal to 1.

The *ncon* parameter specifies the number of vertex weights associated with each vertex of the graph. The value of this parameter determines whether or not METIS will use the multi-constraint partitioning algorithms (Section 3.3). If this parameter is omitted, then the vertices of the graph are assumed to have a single weight. Note that if *ncon* is greater than 0, then the file should contain the required vertex weights and the *fmt* parameter should be set appropriately (i.e., the 2nd bit from right to left should be set to 1).

The remaining n lines store information about the actual structure of the graph. In particular, the ith line (excluding comment lines) contains information that is relevant to the ith vertex. Depending on the value of the *fmt* and *ncon* parameters, the information stored at each line is somewhat different. In the most general form (when *fmt* = `11` and *ncon* > 1) each line will have the following structure (all elements are integer):

$$s\ w_1\ w_2\ \ldots\ w_{ncon}\ v_1\ e_1\ v_2\ e_2\ \ldots\ v_k\ e_k$$

where s is the size of the vertex, $w_1, w_2, \ldots, w_{ncon}$ are the *ncon* vertex weights associated with this vertex, $v_1, \ldots, v_k$ are the vertices adjacent to this vertex, and $e_1, \ldots, e_k$ are the weights of these edges. The vertices (i.e., the various v_i entries) are numbered starting from 1 (not from 0 as is often done in C). Furthermore, the vertex-sizes and vertex-weights must be integers greater or equal to 0, whereas the edge-weights must be strictly greater than 0.

Vertex size versus vertex weights The graph format allows for the specification of both vertex sizes and vertex weights. These two quantities are used by METIS for two entirely different purposes. The vertex weights are used for ensuring that the computed partitionings satisfy the specified balancing constraints (e.g., the sum of the weights of the vertices assigned to each partition is the same across the partitions). On the other hand, the vertex sizes are used for determining the total communication volume, when `gpmetis` and `mpmetis` are invoked with the `-objtype=vol` option. Additional details on how the vertex size is used to determine the communication volume are provided in Section 5.7, which provides the precise formula for computing the total communication volume.

Examples Figure 2 illustrates the format by providing some examples. The simplest format for a graph G is when the size and weight of all vertices and the weight of all the edges is the same. This format is illustrated in Figure 2(a). Note, the optional *fmt* parameter is skipped in this case. However, there are cases in which the edges in G have different weights. This is accommodated as shown in Figure 2(b). Now, the adjacency list of each vertex contains the weight of the edges in addition to the vertices that is connected with. If v has k vertices adjacent to it, then the line for v in the graph file contains $2 * k$ numbers, each pair of numbers stores the vertex that v is connected to, and the weight of the edge. Note that the *fmt* parameter is equal to `001`, indicating the fact that G has weights on the edges. In addition to having weights on the edges, weights on the vertices are also allowed, as illustrated in Figure 2(c). In this case, the value of *fmt* is equal to `011`, and each line of the graph file first stores the weight of the vertex, and then the weighted adjacency list. Finally, Figure 2(d) illustrates the format of the input file when the vertices of the graph contain multiple weights (3 in this example). In this case, the value of *fmt* is equal to `010`, and the value of *ncon* is equal to 3 (since we have three sets of vertex-weights). Each line of the graph file stores the three weights of the vertices followed by the adjacency list.

4.1.2 Mesh file

The primary input of the mesh partitioning programs in METIS is the mesh to be partitioned. This mesh is stored in a file in the form of the element node array. A mesh with n elements is stored in a plain text file that contains $n+1$ lines. The first line (i.e., the *header* line) contains information about the size and the type of the mesh, while the remaining n lines contain the nodes that make up each element.

The first line contains two integer parameters. The first parameter is the number of elements n in the mesh. The second parameter, which is optional, is the number of weights associated with each element. This is equivalent to the *ncon* parameter of the graph file and is used to specify the weights of the vertices in the dual graph. If this parameter is omitted, the weights of the vertices in the dual graph are assumed to be 1[1].

[1]The current mesh format does not allow for the specification of weights with the vertices in the nodal graph. If you need to partition the nodal graph of a mesh whose vertices can have different weights, then the `m2gmetis` routine should be used to first create the nodal graph, which should

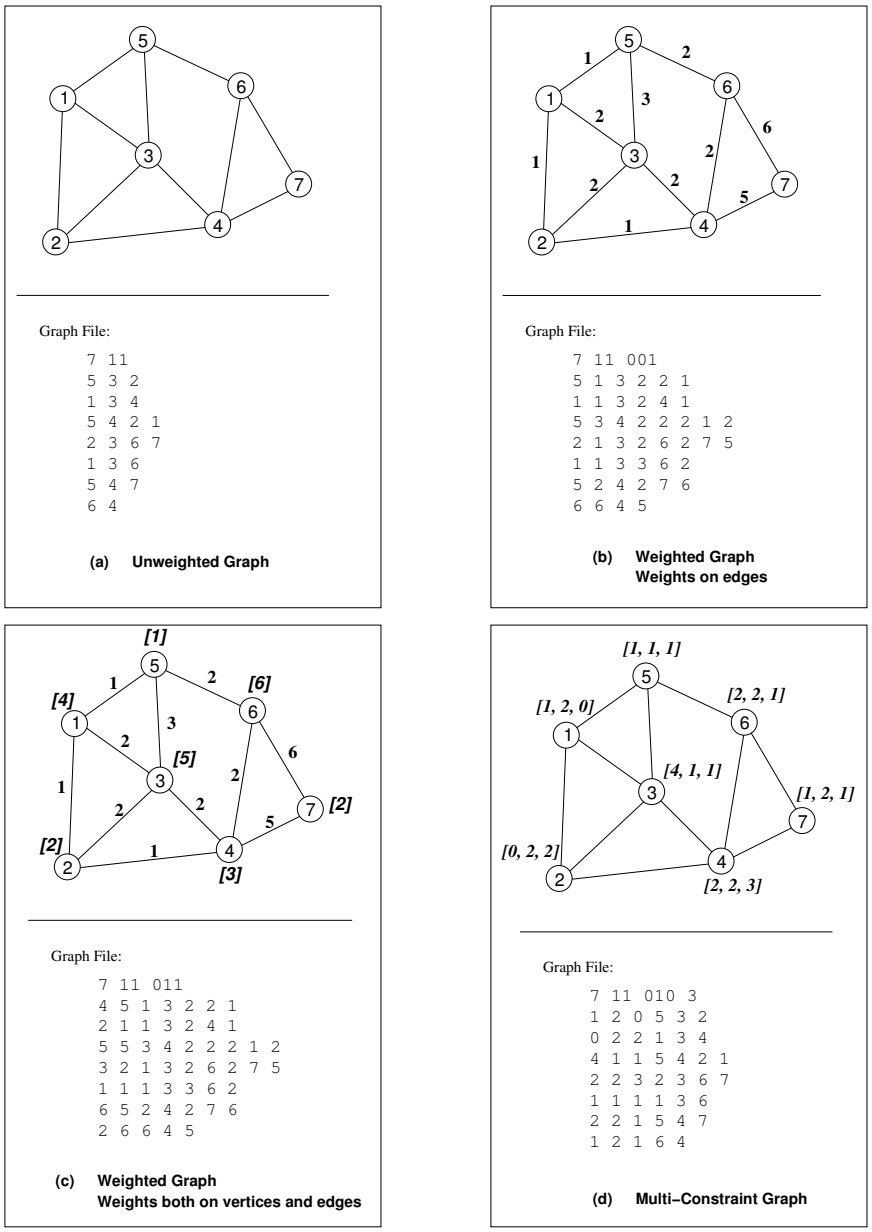


Figure 2: Storage format for various type of graphs.

After the first line, the remaining n lines store the element node array. In particular for element i, line $i + 1$ stores the *ncon* integer weights associated with the ith element (if the optional *ncon* parameter has been specified) followed by the nodes that this element is made off. The weights and nodes are separated by spaces. The numbering of the nodes starts from 1. The nodes of each element can be stored in any order. The mesh file allows for mixed elements meshes, and as such the number of nodes that are supplied in each line can vary.

4.1.3 Target partition weights file

The graph and mesh-partitioning routines take as input an optional file that specifies the target weights of the various partitions (using the `-tpwgts` option. This file contains a sequence of lines, whose format are of the form:

```
frompid [- topid] [: fromcnum [- tocnum]] = twgt
```

be subsequently edited to include the required weights, prior to using `gpmetis` to partition it.

where `frompid` and `topid` specify partition numbers (numbering starting from 0), `fromcnum` and `tocnum` specify constraint numbers (numbering starting from 0), and `twgt` is a floating point number specifying a fraction of the total weight (i.e., better be ≤ 1.0). The parts in square-brackets indicate optional parts. The meaning of the above specification is as follows: For each of the constraints from `fromcnum` up to and including `tocnum` for the partitions starting from `frompid` up to and including `topid` will be assigned a target weight of `twgt`. If `[- topid]` is not supplied, then `topid` = `frompid`. If `[- tocnum]` is not supplied, then `tocnum` = `fromcnum`. If `[: fromcnum [- tocnum]]`, then `fromcnum`=0 and `tocnum`=`ncon`-1.

If the file does not contain a `twgt` specification for all the target partition weight/constraint combinations, then the `twgt` for the unspecified ones is determined by equally distributing the left-over portion of the total weight for each constraint. It is important that for each constraint, the sum of the specified `twgt`s values is less than or equal to 1.0.

For example, assuming that `ncon=1`, `nparts=5`, then the following `tpwgts` file

```
0-1 = .2
4   = .3
```

specifies the following target partition weights:

`part[0]=.2; part[1]=.2; part[2]=.15; part[3]=.15; and part[4]=0.3`

Note that the `.15` fractions for `part[2]` and `part[3]` are due to the equal distribution of the left-over weight (1.0 - .7).

4.2 Output file formats

The output of METIS is either a partition or an ordering file, depending on whether METIS is used for graph/mesh partitioning or for sparse matrix ordering. The format of these files are described in the following sections.

4.2.1 Partition file

The partition file of a graph with n vertices consists of n lines with a single number per line. The ith line of the file contains the partition number that the ith vertex belongs to. Partition numbers start from 0 up to the number of partitions minus one.

4.2.2 Ordering file

The ordering file of a graph with n vertices consists of n lines with a single number per line. The ith line of the ordering file contains the new order of the ith vertex of the graph. The numbering in the ordering file starts from 0.

Note that the ordering file stores what is referred to as the the inverse permutation vector *iperm* of the ordering. Let A be a matrix and let A' be the reordered matrix. The inverse permutation vector maps the ith row (column) of A into the *iperm*$[i]$ row (column) of A'.

4.3 Programs

gpmetis [options] graphfile nparts

Description

Partitions a graph into a specified number of parts. The computed partitioning is stored in a file named `graphfile.part.nparts`, where `graphfile` and `nparts` correspond to the specified parameters.

Parameters

graphfile The name of the file that stores the graph to be partitioned (Section 4.1.1).

nparts The number of parts that the graph will be partitioned into. It should be greater than 1.

Options

-ptype=string

Specifies the scheme used for computing the partitioning. Possible values:

`rb` Multilevel recursive bisectioning.

`kway` Multilevel k-way partitioning [default].

-ctype=string

Specifies the scheme used to match the vertices of the graph during the coarsening. Possible values:

`rm` Random matching.

`shem` Sorted heavy-edge matching [default].

-iptype=string (applies only when -ptype=rb)

Specifies the scheme used to compute the initial partitioning of the graph. Possible values:

`grow` Grows a bisection using a greedy strategy [default].

`random` Computes a bisection at random followed by a refinement.

-objtype=string (applies only when -ptype=kway)

Specifies the partitioning's objective function. Possible values:

`cut` Edgecut minimization [default].

`vol` Total communication volume minimization.

-no2hop Specifies that the coarsening will not perform any 2–hop matchings when the standard matching approach fails to sufficiently coarsen the graph. The 2–hop matching is very effective for graphs with power-law degree distributions. If the quality of the partitionings when compared to that obtained by earlier versions of METIS is low, then this option can be specified to turn off this type of matching.

-contig (applies only when -ptype=kway)

Specifies that the partitioning routines should try to produce partitions that are contiguous. Note that if the input graph is not connected this option is ignored.

-minconn (applies only when -ptype=kway)

Specifies that the partitioning routines should try to minimize the maximum degree of the subdomain graph, i.e., the graph in which each partition is a node, and edges connect subdomains with a shared interface.

-tpwgts=filename

Specifies the name of the file that stores for each constraint the target weights for each partition (Section 4.1.3). By default, for each constraint, all partitions are assumed to be of the same size.

-ufactor=int

Specifies the maximum allowed load imbalance among the partitions. It is this described under `METIS_OPTION_UFACTOR` in Section 5.4.

Note that in the case of multiple constraints, the same load imbalance tolerance is applied to all the constraints. Use -ubvec to provide per-constraint load imbalance tolerances.

-ubvec=string

Specifies the per-constraint allowed load imbalance among partitions. The string must contain a space separated list of floating point numbers, one for each of the constraints. For example, for three constraints, the string can be `"1.02 1.2 1.35"` indicating a desired maximum load imbalance of 2%, 20%, and 35%, respectively. The load imbalance is defined in a way similar to -ufactor. If supplied, this parameter takes priority over ufactor.

-niter=int

Specifies the number of iterations for the refinement algorithms at each stage of the uncoarsening process. Default is 10.

-ncuts=int

Specifies the number of different partitionings that it will compute. The final partitioning is the one that achieves the best edgecut or communication volume. Default is 1.

-nooutput

Specifies that no partitioning file should be generated.

-seed=int Selects the seed of the random number generator.

-dbglvl=int

Specifies the type of progress/debugging information that will be printed to `stdout`. The supplied value corresponds to the addition (bitwise OR) of the various values described in Section 5.4 for `METIS_OPTION_DBGLVL`. The default value is 0 (no progress/debugging information).

-help Displays the command-line options along with a description.

mpmetis [options] meshfile nparts

Description

Partitions a mesh into a specified number of parts. The computed partitioning is stored in two files named: `meshfile.npart.nparts` that stores the partitioning of the nodes, and `meshfile.epart.nparts` that stores the partitioning of the elements. The `meshfile` and `nparts` components of those files correspond to the specified parameters. The format of the partitioning files is described in Section 4.2.1.

Parameters

meshfile The name of the file that stores the mesh to be partitioned (Section 4.1.2).

nparts The number of parts that the mesh will be partitioned into. It should be greater than 1.

Options

-gtype=string

Specifies the graph to be used for computing the partitioning The possible values are:

`dual` Partition the dual graph of the mesh [default].

`nodal` Partition the nodal graph of the mesh.

-ncommon=int (applies only when -gtype=dual)

Specifies the number of common nodes that two elements must have in order to put an edge between them in the dual graph. Given two elements e_1 and e_2, containing n_1 and n_2 nodes, respectively, then an edge will connect the vertices in the dual graph corresponding to e_1 and e_2 if the number of common nodes between them is greater than or equal to $\min(ncommon, n_1 - 1, n_2 - 1)$.

The default value is 1, indicating that two elements will be connected via an edge as long as they share one node. However, this will tend to create too many edges (increasing the memory and time requirements of the partitioning). The user should select higher values that are better suited for the element types of the mesh that wants to partition. For example, for tetrahedron meshes, ncommon should be 3, which creates an edge between two tets when they share a triangular face (i.e., 3 nodes).

-ptype, -ctype, -iptype, -objtype, -contig, -minconn, -tpwgts, -ufactor, -niter, -ncuts, -nooutput, -seed, -dbglvl, -help

Similar to the corresponding options of `gpmetis`.

Notes

The current version of `mpmetis` supports only single constraint partitioning.

ndmetis [options] graphfile

Description

Computes a fill-reducing ordering of the vertices of the graph using multilevel nested dissection. The computed ordering is stored in a file named `graphfile.iperm`, whose format is described in Section 4.2.2.

Parameters

graphfile The name of the file that stores the graph to be re-ordered (Section 4.1.1).

Options

-ctype=string

Specifies the scheme to be used to match the vertices of the graph during the coarsening. The possible values are:

`rm` Random matching [default].

`shem` Sorted heavy-edge matching.

-iptype=string (applies only when -ptype=rb)

Specifies the scheme to be used to compute the initial vertex separator of the graph. The possible values are:

`edge` Derive the separator from an edge cut [default].

`node` Grow a bisection using a greedy node-based strategy.

-rtype=string

Specifies the scheme to be used for refinement. The possible values are:

`1sided` One-sided node-based refinement [default].

`2sided` Two-sided node-based refinement.

-ufactor=int

Specifies the maximum allowed load imbalance between the left and right partitions during each bisection. It is this described under `METIS_OPTION_UFACTOR` in Section 5.4 when the number of partitions . Default is 30, indicating a load imbalance of 1.03.

-pfactor=int

Specifies the minimum degree of the vertices that will be ordered last. It is this described under `METIS_OPTION_PFACTOR` in Section 5.4. Default value is 0, indicating that no vertices are removed.

-no2hop Specifies that the coarsening will not perform any 2–hop matchings when the standard matching approach fails to sufficiently coarsen the graph. The 2–hop matching is very effective for graphs with power-law degree distributions. If the quality of the orderings when compared to that obtained by earlier versions of METIS is low, then this option can be specified to turn off this type of matching.

-nocompress

Specifies that the graph should not be compressed by combining together vertices that have identical adjacency lists.

-ccorder Specifies if the connected components of the graph should first be identified and ordered separately.

-niter=int

Specifies the number of iterations for the refinement algorithms at each stage of the uncoarsening process. Default is 10.

-nseps=int

Specifies the number of different separators that it will compute at each level of nested dissection. The final separator that is used is the smallest one. Default is 1.

-nooutput

Specifies that no ordering file should be generated.

-seed=int Selects the seed of the random number generator.

-dbglvl=int

Similar to the corresponding option of `gpmetis`.

-help Displays the command-line options along with a description.

m2gmetis [options] meshfile graphfile

Description

Converts a mesh into a graph that is compatible with METIS.

Parameters

meshfile The name of the file that stores the mesh to be converted (Section 4.1.2).

graphfile The name of the file that will store the generated graph.

Options

-gtype=string

Specifies the type of the graph to be generated. The possible values are:

`dual` Generates the dual graph of the mesh [default].

`nodal` the nodal graph of the mesh.

-ncommon=int (applies only when -gtype=dual)

Similar to the corresponding option of `mpmetis`.

-dbglvl=int

Similar to the corresponding option of `gpmetis`.

-help Displays the command-line options along with a description.

graphchk graphfile [fixedfile]

Description

Checks the graph for format correctness and consistency.

Parameters

graphfile The name of the file that stores the graph to be checked (Section 4.1.2).

fixedfile This is an optional parameter that specifies the name of the file to store the fixed input graph. This file will only be generated if there were errors in the input file.

5 METIS' API

The various routines implemented in METIS' stand-alone programs can be directly accessed from a C, C++, or Fortran program by using the supplied library. In the rest of this section we describe METIS' API by first describing various calling and usage conventions, the various data structures used to pass information into and get information out of the routines, followed by a detailed description of the calling sequence of the various routines.

5.1 Header files

Any program using METIS' API needs to include the `metis.h` header file. This file provides function prototypes for the various API routines and defines the various data types and constants used by these routines.

During METIS' installation time, the `metis.h` defines two important data types and their widths. These are the `idx_t` data type for storing integer quantities and the `real_t` data type for storing floating point quantities. The `idx_t` data type can be defined to be either a 32 or 64 bit signed integer, whereas the `real_t` data type can be defined to be either a single or double precision float point number. All of METIS' API routines take as input arrays and/or scalars that are of these two data types. In addition, `metis.h` defines various `enum` data types for specifying various options and for returning status codes.

5.2 Use of `NULL` parameters

METIS' API routines take a large number of parameters, allowing the user to model complex graphs and specify complex partitioning/ordering requirements. However, for most uses of METIS, this level of complexity may not be required. For that purpose and to also simplify the complexity associated with using its API, METIS allows the application to specify a `NULL` value to many of these optional/advanced parameters. The API routines that will be described in subsequent sections will mark these parameters by following them with a `(NULL)`.

5.3 C/C++ and Fortran Support

The various routines in METIS' API can be called from either C/C++ or Fortran programs. Using C/C++ with METIS' API is quite straightforward (as METIS is written entirely in C). However, METIS' API fully supports Fortran as well. This support comes in three forms.

1. All the scalar arguments in the routines are passed by reference to facilitate Fortran programs.

2. All the routines take a parameter called *numflag* or an `options` parameter called `METIS_OPTION_NUMBERING` indicating whether or not the numbering of the graph or mesh starts from 0 or 1. In C programs numbering usually starts from 0, whereas in Fortran programs numbering starts from 1.

3. METIS' API incorporates alternative names for each of the routines to facilitate linking the library with Fortran programs. In particular, for every function, METIS' API provides four additional names, one all capital, one all lower case, one all lower case with '_' appended to it, and one with '__' appended to it. For example, for `METIS_PartGraphKway`, METIS' API provides `METIS_PARTGRAPHKWAY`, `metis_partgraphkway`, `metis_partgraphkway_`, and `metis_partgraphkway__`. These extra names allow the library to be directly linked into Fortran programs on a wide range of architectures including Cray, SGI, and HP. If you still encounter problems linking with the library let us know so we can include appropriate support.

5.4 Options array

Most of the API routines take as a parameter an array called `options`, which allow the application to fine-tune and modify various aspects of the internal algorithms used by METIS. The application must define this array as

```
idx_t options[METIS_NOPTIONS];
```

and the meaning of its various entries are as follows:

options[METIS_OPTION_PTYPE]

Specifies the partitioning method. Possible values are:

METIS_PTYPE_RB	Multilevel recursive bisectioning.
METIS_PTYPE_KWAY	Multilevel k-way partitioning.

options[METIS_OPTION_OBJTYPE]

Specifies the type of objective. Possible values are:

METIS_OBJTYPE_CUT	Edge-cut minimization.
METIS_OBJTYPE_VOL	Total communication volume minimization.

options[METIS_OPTION_CTYPE]

Specifies the matching scheme to be used during coarsening. Possible values are:

METIS_CTYPE_RM	Random matching.
METIS_CTYPE_SHEM	Sorted heavy-edge matching.

options[METIS_OPTION_IPTYPE]

Determines the algorithm used during initial partitioning. Possible values are:

METIS_IPTYPE_GROW	Grows a bisection using a greedy strategy.
METIS_IPTYPE_RANDOM	Computes a bisection at random followed by a refinement.
METIS_IPTYPE_EDGE	Derives a separator from an edge cut.
METIS_IPTYPE_NODE	Grow a bisection using a greedy node-based strategy.

options[METIS_OPTION_RTYPE]

Determines the algorithm used for refinement. Possible values are:

METIS_RTYPE_FM	FM-based cut refinement.
METIS_RTYPE_GREEDY	Greedy-based cut and volume refinement.
METIS_RTYPE_SEP2SIDED	Two-sided node FM refinement.
METIS_RTYPE_SEP1SIDED	One-sided node FM refinement.

options[METIS_OPTION_NCUTS]

Specifies the number of different partitionings that it will compute. The final partitioning is the one that achieves the best edgecut or communication volume. Default is 1.

options[METIS_OPTION_NSEPS]

Specifies the number of different separators that it will compute at each level of nested dissection. The final separator that is used is the smallest one. Default is 1.

options[METIS_OPTION_NUMBERING]

Used to indicate which numbering scheme is used for the adjacency structure of a graph or the element-node structure of a mesh. The possible values are:

0	C-style numbering is assumed that starts from 0.
1	Fortran-style numbering is assumed that starts from 1.

options[METIS_OPTION_NITER]

Specifies the number of iterations for the refinement algorithms at each stage of the uncoarsening process. Default is 10.

options[METIS_OPTION_SEED]

Specifies the seed for the random number generator.

options[METIS_OPTION_MINCONN]

Specifies that the partitioning routines should try to minimize the maximum degree of the subdomain graph, i.e., the graph in which each partition is a node, and edges connect subdomains with a shared interface.

0 Does not explicitly minimize the maximum connectivity.

1 Explicitly minimize the maximum connectivity.

options[METIS_OPTION_NO2HOP]

Specifies that the coarsening will not perform any 2–hop matchings when the standard matching approach fails to sufficiently coarsen the graph. The 2–hop matching is very effective for graphs with power-law degree distributions.

0 Performs a 2–hop matching.

1 Does not perform a 2–hop matching.

options[METIS_OPTION_CONTIG]

Specifies that the partitioning routines should try to produce partitions that are contiguous. Note that if the input graph is not connected this option is ignored.

0 Does not force contiguous partitions.

1 Forces contiguous partitions.

options[METIS_OPTION_COMPRESS]

Specifies that the graph should be compressed by combining together vertices that have identical adjacency lists.

0 Does not try to compress the graph.

1 Tries to compress the graph.

options[METIS_OPTION_CCORDER]

Specifies if the connected components of the graph should first be identified and ordered separately.

0 Does not identify the connected components.

1 Identifies the connected components.

options[METIS_OPTION_PFACTOR]

Specifies the minimum degree of the vertices that will be ordered last. If the specified value is $x > 0$, then any vertices with a degree greater than $0.1*x*$(average degree) are removed from the graph, an ordering of the rest of the vertices is computed, and an overall ordering is computed by ordering the removed vertices at the end of the overall ordering. For example if $x = 40$, and the average degree is 5, then the algorithm will remove all vertices with degree greater than 20. The vertices that are removed are ordered last (i.e., they are automatically placed in the top-level separator). Good values are often in the range of 60 to 200 (i.e., 6 to 20 times more than the average). Default value is 0, indicating that no vertices are removed.

Used to control whether or not the ordering algorithm should remove any vertices with high degree (i.e., dense columns). This is particularly helpful for certain classes of LP matrices, in which there a few vertices that are connected to many other vertices. By removing these vertices prior to ordering, the quality and the amount of time required to do the ordering improves.

options[METIS_OPTION_UFACTOR]

Specifies the maximum allowed load imbalance among the partitions. A value of x indicates that the allowed load imbalance is $(1 + x)/1000$. The load imbalance for the jth constraint is defined to be $\max_i(w[j, i])/t[j, i])$, where $w[j, i]$ is the fraction of the overall weight of the jth constraint that is assigned to the ith partition and $t[j, i]$ is the desired target weight of the jth constraint for the ith partition (i.e., that specified via -tpwgts). For -ptype=rb, the default value is 1 (i.e., load imbalance of 1.001) and for -ptype=kway, the default value is 30 (i.e., load imbalance of 1.03).

options[METIS_OPTION_DBGLVL]

Specifies the amount of progress/debugging information will be printed during the execution of the algorithms. The default value is 0 (no debugging/progress information). A non-zero value can be supplied that is obtained by a bit-wise OR of the following values.

METIS_DBG_INFO (1)	Prints various diagnostic messages.
METIS_DBG_TIME (2)	Performs timing analysis.
METIS_DBG_COARSEN (4)	Displays various statistics during coarsening.
METIS_DBG_REFINE (8)	Displays various statistics during refinement.
METIS_DBG_IPART (16)	Displays various statistics during initial partitioning.
METIS_DBG_MOVEINFO (32)	Displays detailed information about vertex moves during refinement.
METIS_DBG_SEPINFO (64)	Displays information about vertex separators.
METIS_DBG_CONNINFO (128)	Displays information related to the minimization of subdomain connectivity.
METIS_DBG_CONTIGINFO (256)	Displays information related to the elimination of connected components.

Note that the numeric values are provided for use with the `-dbglvl` option of METIS' stand-alone programs. For the API routines it is sufficient to OR the above constants.

If an application does not want to take advantage of this capability, then it can just supply a `NULL` as the value for that parameter. For those applications that will like to modify certain elements of the algorithms, METIS provide the `METIS_SetDefaultOptions` routine to set the options to their default values. After that, the application can just modify the the options that is interested in modifying. This is illustrated as follows:

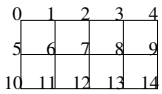
```
idx_t options[METIS_NOPTIONS];

METIS_SetDefaultOptions(options);
options[METIS_OPTION_NSEPS] = 10;
options[METIS_OPTION_UFACTOR] = 100;

METIS_NodeND(..., options, ...)
...
```

5.5 Graph data structure

All of the graph partitioning and sparse matrix ordering routines in METIS take as input the adjacency structure of the graph and the weights of the vertices and edges (if any).

The adjacency structure of the graph is stored using the compressed storage format (CSR). The CSR format is a widely used scheme for storing sparse graphs. In this format the adjacency structure of a graph with n vertices and m edges is represented using two arrays `xadj` and `adjncy`. The `xadj` array is of size $n+1$ whereas the `adjncy` array is of size $2m$ (this is because for each edge between vertices v and u we actually store both (v,u) and (u,v)).

The adjacency structure of the graph is stored as follows. Assuming that vertex numbering starts from 0 (C style), then the adjacency list of vertex i is stored in array `adjncy` starting at index `xadj[`i`]` and ending at (but not including) index `xadj[`$i+1$`]` (i.e., `adjncy[xadj[`i`]]` through and including `adjncy[xadj[`$i+1$`]-1]`). That is, for each vertex i, its adjacency list is stored in consecutive locations in the array `adjncy`, and the array `xadj` is used to point to where it begins and where it ends. Figure 3(b) illustrates the CSR format for the 15-vertex graph shown in Figure 3(a).



(a) A sample graph

xadj	0 2 5 8 11 13 16 20 24 28 31 33 36 39 42 44
adjncy	1 5 0 2 6 1 3 7 2 4 8 3 9 0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14 5 11 6 10 12 7 11 13 8 12 14 9 13

(b CSR format

Figure 3: An example of the CSR format for storing sparse graphs.

The weights of the vertices (if any) are stored in an additional array called `vwgt`. If *ncon* is the number of weights associated with each vertex, the array `vwgt` contains $n * ncon$ elements (recall that n is the number of vertices). The weights of the ith vertex are stored in *ncon* consecutive entries starting at location `vwgt[`$i * ncon$`]`. Note that if each vertex has only a single weight, then `vwgt` will contain n elements, and `vwgt[`i`]` will store the weight of the ith vertex. The vertex-weights must be integers greater or equal to zero. If all the vertices of the graph have the same weight (i.e., the graph is unweighted), then the `vwgt` can be set to NULL.

The weights of the edges (if any) are stored in an additional array called `adjwgt`. This array contains $2m$ elements, and the weight of edge `adjncy[`j`]` is stored at location `adjwgt[`j`]`. The edge-weights must be integers greater than zero. If all the edges of the graph have the same weight (i.e., the graph is unweighted), then the `adjwgt` can be set to NULL.

5.6 Mesh data structure

All of the mesh partitioning and mesh conversion routines in METIS take as input the element node array of a mesh. This element node array is stored using a pair of arrays called `eptr` and `eind`, which are similar to the `xadj` and `adjncy` arrays used for storing the adjacency structure of a graph. The size of the `eptr` array is $n+1$, where n is the number of elements in the mesh. The size of the `eind` array is of size equal to the sum of the number of nodes in all the elements of the mesh. The list of nodes belonging to the ith element of the mesh are stored in consecutive locations of `eind` starting at position `eptr[i]` up to (but not including) position `eptr[i+1]`. This format makes it easy to specify meshes of any type of elements, including meshes with mixed element types that have different number of nodes per element. As it was the case with the format of the mesh file described in Section 4.1.2, the ordering of the nodes in each element is not important.

5.7 Partitioning objectives

The partitioning algorithms in METIS can be used to compute a balanced k-way partitioning that minimizes either the number of edges that straddle partitions (*edgecut*) or the total communication volume (*totalv*). In the rest of this section we briefly describe these two objectives and provide some suggestions on when they should be used.

Minimizing the edgecut Consider a graph $G = (V, E)$, and let P be a vector of size $|V|$ such that $P[i]$ stores the number of the partition that vertex i belongs to. The *edgecut* of this partitioning is defined as the number of edges that straddle partitions. That is, the number of edges (v, u) for which $P[v] \neq P[u]$. If the graph has weights associated with the edges, then the edgecut is defined as the sum of the weight of these straddling edges.

Minimizing the total communication volume Consider a graph $G = (V, E)$, and let P be a vector of size $|V|$ such that $P[i]$ stores the number of the partition that vertex i belongs to. Let $V_b \subset V$ be the subset of interface (or boarder) vertices. That is, each vertex $v \in V_b$ is connected to at least one vertex that belongs to a different partition. For each vertex $v \in V_b$ let $Nadj[v]$ be the number of domains other than $P[v]$ that the vertices adjacent to v belong to.

The *totalv* of this partitioning is defined as:

$$\textit{totalv} = \sum_{v \in V_b} Nadj[v]. \tag{1}$$

Equation 1 corresponds to the total communication volume incurred by the partitioning because each interface vertex v needs to be sent to all of its $Nadj[v]$ partitions.

The above model can be extended to instances in which the amount of data that needs to be sent for each node is different. In particular, if s_v is the amount of data that needs to be sent for vertex v, referred to as the *vertex size*, then Equation 1 can be re-written as:

$$\textit{totalv} = \sum_{v \in V_b} s_v Nadj[v]. \tag{2}$$

METIS' API supports this weighted totalv model by using an array called `vsize` such that the amount of data that needs to be sent due to the ith vertex is stored in `vsize[`i`]`. Note that the amount of data that needs to be sent is different from the *weight* of the vertex. The former corresponds to communication cost whereas the later corresponds to the computational cost.

Note that for partitioning algorithms to correctly minimize the totalv, the graph should reflect the true information exchange requirements of the underlying computations. For instance, the dual graph of a finite element mesh does not correctly model the underlying communication, whereas the nodal graph does.

Which one is better? When partitioning is used to distribute a graph or a mesh among the processors of a parallel computer, the edgecut is only an approximation of the true communication cost resulting from the partitioning. On the other hand, by minimizing the totalv we can directly minimize the overall communication cost. Despite of that, for many graphs the solutions obtained by minimizing the edgecut or minimizing the totalv, are comparable. This is especially true for graphs corresponding to well-shaped finite element meshes. This is because for these graphs, the degrees of the various vertices are similar and the objectives of minimizing the edgecut or the totalv behave the same. On the other hand, if the vertex degrees vary significantly (e.g., graphs corresponding to linear programming matrices), then by minimizing the totalv we can obtain a significant reduction in the total communication volume.

In terms of the amount of time required by these two partitioning objectives, minimizing the edgecut is faster than minimizing the totalv. For this reason, the totalv objective should be used only for problems in which it actually reduces the overall communication volume.

5.8 Graph partitioning routines

`int` **METIS_PartGraphRecursive**(`idx_t` *nvtxs, `idx_t` *ncon, `idx_t` *xadj, `idx_t` *adjncy, `idx_t` *vwgt, `idx_t` *vsize, `idx_t` *adjwgt, `idx_t` *nparts, `real_t` *tpwgts, `real_t` ubvec, `idx_t` *options, `idx_t` *objval, `idx_t` *part)

`int` **METIS_PartGraphKway**(`idx_t` *nvtxs, `idx_t` *ncon, `idx_t` *xadj, `idx_t` *adjncy, `idx_t` *vwgt, `idx_t` *vsize, `idx_t` *adjwgt, `idx_t` *nparts, `real_t` *tpwgts, `real_t` ubvec, `idx_t` *options, `idx_t` *objval, `idx_t` *part)

Description

Is used to partition a graph into k parts using either multilevel recursive bisection or multilevel k-way partitioning.

Parameters

nvtxs The number of vertices in the graph.

ncon The number of balancing constraints. It should be at least 1.

xadj, adjncy
The adjacency structure of the graph as described in Section 5.5.

vwgt (NULL)
The weights of the vertices as described in Section 5.5.

vsize (NULL)
The size of the vertices for computing the total communication volume as described in Section 5.7.

adjwgt (NULL)
The weights of the edges as described in Section 5.5.

nparts The number of parts to partition the graph.

tpwgts (NULL)
This is an array of size *nparts*×*ncon* that specifies the desired weight for each partition and constraint. The *target partition weight* for the ith partition and jth constraint is specified at `tpwgts[i*ncon+j]` (the numbering for both partitions and constraints starts from 0). For each constraint, the sum of the `tpwgts[]` entries must be 1.0 (i.e., $\sum_i tpwgts[i * ncon + j] = 1.0$).

A `NULL` value can be passed to indicate that the graph should be equally divided among the partitions.

ubvec (NULL)
This is an array of size *ncon* that specifies the allowed load imbalance tolerance for each constraint. For the ith partition and jth constraint the allowed weight is the ubvec[j]*tpwgts[i*ncon+j] fraction of the jth's constraint total weight. The load imbalances must be greater than 1.0.

A `NULL` value can be passed indicating that the load imbalance tolerance for each constraint should be 1.001 (for ncon=1) or 1.01 (for ncon¿1).

options (NULL)
This is the array of options as described in Section 5.4.

The following options are valid for `METIS_PartGraphRecursive`:

```
METIS_OPTION_CTYPE, METIS_OPTION_IPTYPE, METIS_OPTION_RTYPE,
METIS_OPTION_NO2HOP, METIS_OPTION_NCUTS, METIS_OPTION_NITER,
METIS_OPTION_SEED, METIS_OPTION_UFACTOR, METIS_OPTION_NUMBERING,
METIS_OPTION_DBGLVL
```

The following options are valid for `METIS_PartGraphKway`:

METIS_OPTION_OBJTYPE, METIS_OPTION_CTYPE, METIS_OPTION_IPTYPE,
METIS_OPTION_RTYPE, METIS_OPTION_NO2HOP, METIS_OPTION_NCUTS,
METIS_OPTION_NITER, METIS_OPTION_UFACTOR, METIS_OPTION_MINCONN,
METIS_OPTION_CONTIG, METIS_OPTION_SEED, METIS_OPTION_NUMBERING,
METIS_OPTION_DBGLVL

objval Upon successful completion, this variable stores the edge-cut or the total communication volume of the partitioning solution. The value returned depends on the partitioning's objective function.

part This is a vector of size *nvtxs* that upon successful completion stores the partition vector of the graph. The numbering of this vector starts from either 0 or 1, depending on the value of `options[METIS_OPTION_NUMBERING]`.

Returns

`METIS_OK`	Indicates that the function returned normally.
`METIS_ERROR_INPUT`	Indicates an input error.
`METIS_ERROR_MEMORY`	Indicates that it could not allocate the required memory.
`METIS_ERROR`	Indicates some other type of error.

5.9 Mesh partitioning routines

`int` **METIS_PartMeshDual**(`idx_t` *ne, `idx_t` *nn, `idx_t` *eptr, `idx_t` *eind, `idx_t` *vwgt, `idx_t` *vsize, `idx_t` *ncommon, `idx_t` *nparts, `real_t` *tpwgts, `idx_t` *options, `idx_t` *objval, `idx_t` *epart, `idx_t` *npart)

Description

This function is used to partition a mesh into k parts based on a partitioning of the mesh's dual graph.

Parameters

ne The number of elements in the mesh.

nn The number of nodes in the mesh.

eptr, eind

The pair of arrays storing the mesh as described in Section 5.6.

vwgt (NULL)

An array of size *ne* specifying the weights of the elements. A `NULL` value can be passed to indicate that all elements have an equal weight.

vsize (NULL)

An array of size *ne* specifying the size of the elements that is used for computing the total communication volume as described in Section 5.7. A `NULL` value can be passed when the objective is cut or when all elements have an equal size.

ncommon

Specifies the number of common nodes that two elements must have in order to put an edge between them in the dual graph. Given two elements e_1 and e_2, containing n_1 and n_2 nodes, respectively, then an edge will connect the vertices in the dual graph corresponding to e_1 and e_2 if the number of common nodes between them is greater than or equal to $\min(ncommon, n_1 - 1, n_2 - 1)$.

The default value is 1, indicating that two elements will be connected via an edge as long as they share one node. However, this will tend to create too many edges (increasing the memory and time requirements of the partitioning). The user should select higher values that are better suited for the element types of the mesh that wants to partition. For example, for tetrahedron meshes, ncommon should be 3, which creates an edge between two tets when they share a triangular face (i.e., 3 nodes).

nparts The number of parts to partition the mesh.

tpwgts (NULL)

This is an array of size *nparts* that specifies the desired weight for each partition. The *target partition weight* for the ith partition is specified at `tpwgts[i]` (the numbering for the partitions starts from 0). The sum of the `tpwgts[]` entries must be 1.0.

A `NULL` value can be passed to indicate that the graph should be equally divided among the partitions.

options (NULL)

This is the array of options as described in Section 5.4. The following options are valid:

```
METIS_OPTION_PTYPE, METIS_OPTION_OBJTYPE, METIS_OPTION_CTYPE,
METIS_OPTION_IPTYPE, METIS_OPTION_RTYPE, METIS_OPTION_NCUTS,
METIS_OPTION_NITER, METIS_OPTION_SEED, METIS_OPTION_UFACTOR,
METIS_OPTION_NUMBERING, METIS_OPTION_DBGLVL
```

objval Upon successful completion, this variable stores either the edgecut or the total communication volume of the dual graph's partitioning.

epart This is a vector of size *ne* that upon successful completion stores the partition vector for the elements of the mesh. The numbering of this vector starts from either 0 or 1, depending on the value of `options[METIS_OPTION_NUMBERING]`.

npart This is a vector of size *nn* that upon successful completion stores the partition vector for the nodes of the mesh. The numbering of this vector starts from either 0 or 1, depending on the value of `options[METIS_OPTION_NUMBERING]`.

Returns

`METIS_OK`	Indicates that the function returned normally.
`METIS_ERROR_INPUT`	Indicates an input error.
`METIS_ERROR_MEMORY`	Indicates that it could not allocate the required memory.
`METIS_ERROR`	Indicates some other type of error.

int **METIS_PartMeshNodal**(idx_t *ne, idx_t *nn, idx_t *eptr, idx_t *eind, idx_t *vwgt, idx_t *vsize, idx_t *nparts, real_t *tpwgts, idx_t *options, idx_t *objval, idx_t *epart, idx_t *npart)

Description

This function is used to partition a mesh into k parts based on a partitioning of the mesh's nodal graph.

Parameters

ne The number of elements in the mesh.

nn The number of nodes in the mesh.

eptr, eind

The pair of arrays storing the mesh as described in Section 5.6.

vwgt (NULL)

An array of size *nn* specifying the weights of the nodes. A NULL value can be passed to indicate that all nodes have an equal weight.

vsize (NULL)

An array of size *nn* specifying the size of the nodes that is used for computing the total communication volume as described in Section 5.7. A NULL value can be passed when the objective is cut or when all nodes have an equal size.

nparts The number of parts to partition the mesh.

tpwgts (NULL)

This is an array of size *nparts* that specifies the desired weight for each partition. The *target partition weight* for the ith partition is specified at tpwgts[i] (the numbering for the partitions starts from 0). The sum of the tpwgts[] entries must be 1.0.

A NULL value can be passed to indicate that the graph should be equally divided among the partitions.

options (NULL)

This is the array of options as described in Section 5.4. The following options are valid:

```
METIS_OPTION_PTYPE, METIS_OPTION_OBJTYPE, METIS_OPTION_CTYPE,
METIS_OPTION_IPTYPE, METIS_OPTION_RTYPE, METIS_OPTION_NCUTS,
METIS_OPTION_NITER, METIS_OPTION_SEED, METIS_OPTION_UFACTOR,
METIS_OPTION_NUMBERING, METIS_OPTION_DBGLVL
```

objval Upon successful completion, this variable stores either the edgecut or the total communication volume of the nodal graph's partitioning.

epart This is a vector of size *ne* that upon successful completion stores the partition vector for the elements of the mesh. The numbering of this vector starts from either 0 or 1, depending on the value of options[METIS_OPTION_NUMBERING].

npart This is a vector of size *nn* that upon successful completion stores the partition vector for the nodes of the mesh. The numbering of this vector starts from either 0 or 1, depending on the value of options[METIS_OPTION_NUMBERING].

Returns

METIS_OK	Indicates that the function returned normally.
METIS_ERROR_INPUT	Indicates an input error.
METIS_ERROR_MEMORY	Indicates that it could not allocate the required memory.
METIS_ERROR	Indicates some other type of error.

5.10 Sparse Matrix Reordering Routines

`int` **METIS_NodeND**(`idx_t` *nvtxs, `idx_t` *xadj, `idx_t` *adjncy, `idx_t` *vwgt, `idx_t` *options, `idx_t` *perm, `idx_t` *iperm)

Description

This function computes fill reducing orderings of sparse matrices using the multilevel nested dissection algorithm.

Parameters

nvtxs The number of vertices in the graph.

xadj, adjncy

The adjacency structure of the graph as described in Section 5.5.

vwgt (NULL)

An array of size *nvtxs* specifying the weights of the vertices. If the graph is weighted, the nested dissection ordering computes vertex separators that minimize the sum of the weights of the vertices on the separators.

A `NULL` can be passed to indicate a graph with equal weight vertices (or unweighted).

options (NULL)

This is the array of options as described in Section 5.4. The following options are valid:

```
METIS_OPTION_CTYPE, METIS_OPTION_RTYPE, METIS_OPTION_NO2HOP,
METIS_OPTION_NSEPS, METIS_OPTION_NITER, METIS_OPTION_UFACTOR,
METIS_OPTION_COMPRESS, METIS_OPTION_CCORDER, METIS_OPTION_SEED,
METIS_OPTION_PFACTOR, METIS_OPTION_NUMBERING, METIS_OPTION_DBGLVL
```

perm, iperm

These are vectors, each of size *nvtxs*. Upon successful completion, they store the fill-reducing permutation and inverse-permutation. Let A be the original matrix and A' be the permuted matrix. The arrays *perm* and *iperm* are defined as follows. Row (column) i of A' is the perm$[i]$ row (column) of A, and row (column) i of A is the iperm$[i]$ row (column) of A'. The numbering of this vector starts from either 0 or 1, depending on the value of `options[METIS_OPTION_NUMBERING]`.

Returns

`METIS_OK`	Indicates that the function returned normally.
`METIS_ERROR_INPUT`	Indicates an input error.
`METIS_ERROR_MEMORY`	Indicates that it could not allocate the required memory.
`METIS_ERROR`	Indicates some other type of error.

5.11 Mesh-to-graph conversion routines

`int` **METIS_MeshToDual**(`idx_t` *ne, `idx_t` *nn, `idx_t` *eptr, `idx_t` *eind, `idx_t` *ncommon, `idx_t` *numflag, `idx_t` **xadj, `idx_t` **adjncy)

Description

This function is used to generate the dual graph of a mesh.

Parameters

ne The number of elements in the mesh.

nn The number of nodes in the mesh.

eptr, eind

The pair of arrays storing the mesh as described in Section 5.6.

ncommon

Specifies the number of common nodes that two elements must have in order to put an edge between them in the dual graph. Given two elements e_1 and e_2, containing n_1 and n_2 nodes, respectively, then an edge will connect the vertices in the dual graph corresponding to e_1 and e_2 if the number of common nodes between them is greater than or equal to $\min(ncommon, n_1 - 1, n_2 - 1)$.

The default value is 1, indicating that two elements will be connected via an edge as long as they share one node. However, this will tend to create too many edges (increasing the memory and time requirements of the partitioning). The user should select higher values that are better suited for the element types of the mesh that wants to partition. For example, for tetrahedron meshes, ncommon should be 3, which creates an edge between two tets when they share a triangular face (i.e., 3 nodes).

numflag Used to indicate which numbering scheme is used for *eptr* and *eind*. The possible values are:

0 C-style numbering is assumed that starts from 0

1 Fortran-style numbering is assumed that starts from 1

xadj, adjncy

These arrays store the adjacency structure of the generated dual graph. The format is of the adjacency structure is described in Section 5.5. Memory for these arrays is allocated by METIS' API using the standard `malloc` function. It is the responsibility of the application to free this memory by calling `free`. METIS provides the `METIS_Free` that is a wrapper to C's `free` function.

Returns

`METIS_OK`	Indicates that the function returned normally.
`METIS_ERROR_INPUT`	Indicates an input error.
`METIS_ERROR_MEMORY`	Indicates that it could not allocate the required memory.
`METIS_ERROR`	Indicates some other type of error.

`int` **METIS_MeshToNodal**(`idx_t` *ne, `idx_t` *nn, `idx_t` *eptr, `idx_t` *eind, `idx_t` *numflag, `idx_t` **xadj, `idx_t` **adjncy)

Description

This function is used to generate the nodal graph of a mesh.

Parameters

ne The number of elements in the mesh.

nn The number of nodes in the mesh.

eptr, eind

The pair of arrays storing the mesh as described in Section 5.6.

numflag Used to indicate which numbering scheme is used for *eptr* and *eind*. The possible values are:

0 C-style numbering is assumed that starts from 0

1 Fortran-style numbering is assumed that starts from 1

xadj, adjncy

These arrays store the adjacency structure of the generated graph. The format is of the adjacency structure is described in Section 5.5. Memory for these arrays is allocated by METIS' API using the standard `malloc` function. It is the responsibility of the application to free this memory by calling `free`. METIS provides the `METIS_Free` that is a wrapper to C's `free` function.

Returns

`METIS_OK`	Indicates that the function returned normally.
`METIS_ERROR_INPUT`	Indicates an input error.
`METIS_ERROR_MEMORY`	Indicates that it could not allocate the required memory.
`METIS_ERROR`	Indicates some other type of error.

5.12 Utility routines

`int` **METIS_SetDefaultOptions**(`idx_t` options[`METIS_NOPTIONS`])

Description

Initializes the options array into its default values.

Parameters

options The array of options that will be initialized. It's size should be at least `METIS_NOPTIONS`.

Returns

`METIS_OK` Indicates that the function returned normally.

`int` **METIS_Free**(`idx_t` *ptr)

Description

Frees the memory that was allocated by either the `METIS_MeshToDual` or the `METIS_MeshToNodal` routines for returning the dual or nodal graph of a mesh.

Parameters

ptr The pointer to be freed. This pointer should be one of the `xadj` or `adjncy` returned by METIS' API routines.

Returns

`METIS_OK` Indicates that the function returned normally.

6 System requirements and contact information

METIS is written entirely in ANSI C, and is portable on most Unix systems that have an ANSI C compiler (the GNU C compiler will do). It has been tested on Linux, SunOS, and OSX. Instructions on how to build and install METIS can be found in the file `Install.txt` of the distribution.

METIS have been extensively tested on a number of different architectures. However, even though METIS contains no known bugs, this does not mean that all of its bugs have been found and fixed. If you have any problems, please send email to karypis@cs.umn.edu with a brief description of the problem. Also, any future updates to METIS will be made available on WWW at http://www.cs.umn.edu/~metis.

7 Copyright & license notice

References

[1] Bruce Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *Proc. of Irregular 1998*, 1998.

[2] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of Supercomputing*, 1998.

[3] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[4] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[5] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.