juliacon

GraphLab.jl: A Julia Framework for Graph Partitioning

Malik Lechekhab[1], Dimosthenis Pasadakis[1], Roger Käppeli[2], Aryan Eftekhari[1], and Olaf Schenk[1]

[1]Faculty of Informatics, Università della Svizzera italiana (USI), Lugano, Switzerland
[2]Seminar for Applied Mathematics, ETH Zürich

ABSTRACT

We design and implement `GraphLab.jl`, a Julia package that facilitates the study, experimentation, and research of graph partitioning. `GraphLab.jl` provides a framework for exploring the principles and trade-offs of partitioning algorithms through hands-on tools. It offers a growing set of methods—including coordinate, inertial, and spectral bisection, random spheres, space-filling curves, and nested dissection—with support for recursive partitioning. The package includes routines for generating adjacency matrices, computing partition quality metrics, benchmarking problems, and visualizing partitioned graphs. `GraphLab.jl` enables integration with external graph partitioning software, thus allowing users to compare additional methods and results in a unified environment. Last, our work also aims to introduce Julia's capabilities to learners and researchers engaging in graph theory and related partitioning problems.

Keywords

Julia, Graph partitioning, Clustering, Numerical computing

1. Introduction

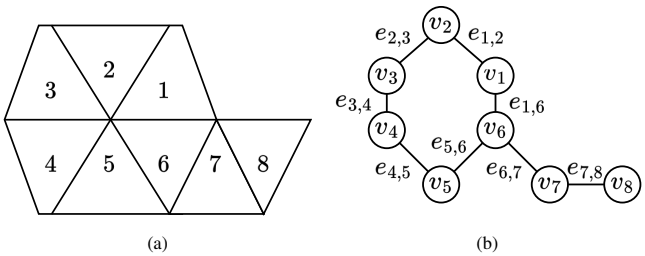
Graph partitioning is a fundamental problem with wide-ranging applications in computational biology, social network analysis, high-performance computing (HPC), and distributed systems. Partitioning large graphs into loosely connected subsets of roughly equal size promotes parallel execution, reduces communication overhead, and provides insights into the structure of complex networks [4].

We contribute to the `Julia` ecosystem of graph algorithms with `GraphLab.jl`, a package designed to facilitate the study, experimentation, and research of graph partitioning. Over the years, a plethora of graph partitioning techniques have been developed. Evaluating their effectiveness and comparing their trade-offs are key to building intuition and practical understanding. `GraphLab.jl` offers a framework that enables users to experiment with algorithms, visualize results, and assess partition quality. The package implements a diverse set of partitioning algorithms, including coordinate [15], inertial [6], and spectral bisection [7, 15], random spheres [9], space-filling curves [13], and nested dissection [8]. These methods can be applied recursively for hierarchical partitioning or multi-level strategies. `GraphLab.jl` also provides routines for generating adjacency matrices, computing partition quality metrics, benchmarking problems, and visualizing partitioned graphs. It allows integration with external graph partitioning software, thus enabling users to compare additional methods and results in a unified environment.

The paper is structured as follows. Section 2 provides background on graph partitioning, followed by an overview the fundamental implemented partitioning algorithms in Section 3. Section 4 introduces our framework, detailing its capabilities in graph creation, benchmarking, visualization, and external software integration. Installation and usage demonstrations are discussed in Section 5, and we conclude with a summary and directions for future work in Section 6.

2. Background on graph partitioning

Consider a mesh consisting of eight cells, as illustrated in Figure 1a. If data exchange occurs only between adjacent cells, the mesh can be represented as a dependency graph, shown in Figure 1b. To partition the mesh into two domains suitable for parallel processing, the objective is to divide it into two submeshes of equal size while minimizing the number of edges connecting them. This corresponds to partitioning the original graph into two complementary subgraphs of equal number of vertices and with a minimum number of interconnecting edges.



(a) (b)

Fig. 1: Mesh example consisting of 8 parts (left) and the corresponding dependency graph (right).

However, for this bisection problem, finding an optimal solution is NP-hard, making exact computation intractable for large instances [3]. In the following, we formalize the graph partitioning problem and introduce bisection algorithms. Here, *bisection* specifically refers partitioning the graph into two sub-graphs. A general partitioning into $p = 2^l$ sub-graphs can then be obtained recursively by applying these bisection methods iteratively.

Let $G = (\mathcal{V}, \mathcal{E})$ be an undirected graph with a vertex set $\mathcal{V} = \{v_1, \ldots, v_n\}$ where each vertex v_i represents an element or entity in the problem domain, and an edge set $\mathcal{E}$ where each edge $e_{i,j} \in \mathcal{E}$ represents a symmetric relation between two distinct vertices v_i and v_j, meaning that $e_{i,j} \in \mathcal{E}$ implies $e_{j,i} \in \mathcal{E}$, and no self-loops exist, i.e., $e_{i,i} \notin \mathcal{E}$. Graphs satisfying these properties are formally referred to as simple and undirected.

The adjacency matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ of the graph stores the connectivity information among its vertices, where the entry $\mathbf{A}_{ij}$ is defined as:

$$\mathbf{A}_{ij} = \begin{cases} a_{ij}, & \text{if } e_{i,j} \in \mathcal{E}, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Here, a_{ij} represents the weight of the edge $e_{i,j}$, which is a nonnegative real-valued number indicating the strength or capacity of the connection between vertices v_i and v_j. In unweighted graphs, a_{ij} simplifies to 1 for all connected pairs. For the simple undirected graphs considered here,the adjacency matrix is symmetric, satisfying $a_{ij} = a_{ji}$ with a zero diagonal, i.e., $a_{ii} = 0$. The degree of a vertex v_i, denoted as $d_i = \sum_j^n a_{ij}$, represents the sum of the weights of edges incident to v_i. In unweighted graphs, d_i reduces to the number of edges connecting v_i, effectively counting its direct neighbors. The degree matrix $\mathbf{D} \in \mathbb{R}^{n \times n}$ is defined as a diagonal matrix, where the diagonal entries correspond to the degrees of all vertices $d_1, \ldots, d_n$.
As an example, for the mesh and the corresponding graph depicted in Figure 1, the adjacency matrix $\mathbf{A}$ and the degree matrix $\mathbf{D}$ are given as follows:

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \mathbf{D} = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

We refer the reader to [1] for a detailed overview of commonly used matrices and objectives functions in graph partitioning.

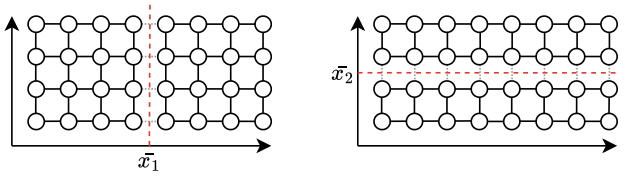
3. Overview of implemented partitioning algorithms

This section provides an overview of the partitioning algorithms implemented in the framework, detailing their underlying principles and computational properties. Through illustrative examples and visual representations, we highlight the behavior of each algorithm.

3.1 Geometric-based partitioning algorithms

This class of bisection algorithms operates under the assumption that the geometric layout of the graph is known. These algorithms exploit spatial information of the vertices to guide the partitioning process, aiming to minimize edge cuts while preserving geometric coherence[9]. This approach is particularly well suited for applications where the graph structure arises from physical systems, such as finite element meshes in numerical computing, where the underlying geometry directly influences computational efficiency[3]. Unless stated otherwise, all function calls presented in this section take as input a graph adjacency matrix `A` and a node coordinate matrix `coords`, and return a vector assigning each node to partition 1 or 2.

3.1.1 Coordinate bisection. Coordinate bisection seeks a hyperplane orthogonal to one of the coordinate axes that partitions the graph's vertices into two subsets of approximately equal size while minimizing the edge cut. The algorithm computes the median $\bar{x}_j$ of each coordinate x_j, dividing all graph vertices into two groups: one containing vertices with $x_j \leq \bar{x}_j$ and the other $x_j > \bar{x}_j$. The edge cut is then evaluated for each coordinate axis, and the partitioning is performed along the axis that yields the smallest edge cut. This process in a d-dimensional space is summarized in Algorithm 1, with an example illustrated in Figure 2.



(a) (b)

Fig. 2: Bisection of the graph along the x_1-axis, resulting in a 4-edge cut (left), and along the x_2-axis, resulting in 8-edge cut (right). The x_1-axis bisection is selected.

Algorithm 1 Coordinate bisection.

Require: Graph $G = (\mathcal{V}, \mathcal{E})$, points $P_i = (x_1, \ldots, x_d)_i$
Ensure: A bisection of G into $\mathcal{V}_1$ and $\mathcal{V}_2$
1: **function** COORDINATE_PART(graph G, points P_i)
2: **for** each axis x_j where $j = 1, \ldots, d$ **do**
3: Compute the median $\bar{x_j}$
4: Compute the edge cut for the bisection at $\bar{x}_i$
5: **end for**
6: Select the axis x_j^* with the smallest median edge cut
7: Partition $\mathcal{V}$ into $\mathcal{V}_1$ and $\mathcal{V}_2$ via $\bar{x_j^*}$ bisection
8: **return** $\mathcal{V}_1, \mathcal{V}_2$
9: **end function**

The coordinate bisection method in `GraphLab.jl` can be invoked using the following command:

```
GraphLab.part_coordinate(A, coords)
```

The coordinate bisection algorithm is computationally efficient and conceptually simple. However, its effectiveness is strongly influenced by the choice of coordinate system. A mere rotation of the coordinate axes can lead to significantly different partitioning results, as the algorithm strictly aligns the division with the coordinate axes. This sensitivity may lead to suboptimal partitions, particularly in cases where the problem geometry is not naturally aligned with the axes, highlighting a fundamental limitation of the algorithm in certain applications.

3.1.2 Inertial bisection. The inertial bisection mitigates the axis-alignment limitation of coordinate bisection by allowing the dividing hyperplane to be orthogonal to a direction determined by the distribution of vertices rather than a fixed coordinate axis. This approach ensures that partitioning is guided by the intrinsic geometry of the data rather than an arbitrary reference frame. In two dimensions, the dividing hyperplane is represented by a line l that minimizes the sum of squared distances from the vertices to the line. The algorithm first determines the center of mass of the vertex set,

$$\bar{P} = (\bar{x}, \bar{y}), \quad \text{where} \quad \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i. \tag{2}$$

It then defines a unit direction vector $\mathbf{u} = [u_1, u_2]^T$, such that $\|\mathbf{u}\|_2 = \sqrt{u_1^2 + u_2^2} = 1$. The parametric equation of the bisecting line is given by $l(\lambda) = \{\bar{P} + \lambda\mathbf{u} \mid \lambda \in \mathbb{R}\}$.

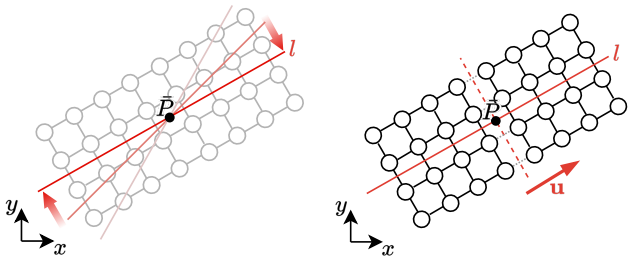


Fig. 3: Illustration of the inertial bisection in 2D: dividing the graph along the line orthogonal to the direction $\mathbf{u}$ through the center of mass $\bar{P}$ that minimizes the sum of squared distances.

To determine the optimal orientation of the bisecting hyperplane, the unit direction vector $\mathbf{u}$ is chosen to minimize the sum of the squared distances from the vertices to the line:

$$\begin{aligned}\sum_{i=1}^{n} d_i^2 &= \sum_{i=1}^{n}(x_i - \bar{x})^2 + (y_i - \bar{y})^2 - (u_1(x_i - \bar{x}) + u_2(y_i - \bar{y}))^2 \\ &= (1 - u_1^2)\sum_{i=1}^{n}(x_i - \bar{x})^2 + (1 - u_2^2)\sum_{i=1}^{n}(y_i - \bar{y})^2 \\ &+ 2u_1u_2\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \\ &= (1 - u_1^2)S_{xx} + (1 - u_2^2)S_{yy} + 2u_1u_2S_{xy} \\ &= \mathbf{u}^T \begin{pmatrix} S_{xx} & S_{xy} \\ S_{xy} & S_{yy} \end{pmatrix} \mathbf{u} = \mathbf{u}^T\mathbf{M}\mathbf{u}\end{aligned}$$

Here, $\mathbf{M}$ is a symmetric matrix, and its smallest eigenvalue corresponds to the minimal sum of the squared distances. Consequently, the optimal direction vector $\mathbf{u}$ is given by the normalized eigenvector associated with the smallest eigenvalue of $\mathbf{M}$ [5]. This choice ensures that the partitioning hyperplane is aligned with the principal axis of least variance, making the algorithm robust to coordinate system transformations. The full procedure for bisecting a graph using inertial partitioning is summarized in Algorithm 2.

Algorithm 2 Inertial bisection.

Require: Graph $G = (\mathcal{V}, \mathcal{E})$, points $P_i = (x_1, \ldots, x_d)_i$
Ensure: A bisection of G into $\mathcal{V}_1$ and $\mathcal{V}_2$
1: **function** INERTIAL_PART(graph G, points P_i)
2: Calculate the center of mass $\bar{P}$
3: Compute eigenvec. associated with smallest eigenval. of $\mathbf{M}$
4: Partition the vertices $\mathcal{V}$ around the line l
5: **return** $\mathcal{V}_1, \mathcal{V}_2$
6: **end function**

To perform inertial bisection with `GraphLab.jl` on a graph, use:

```
GraphLab.part_inertial(A, coords)
```

3.1.3 Random sphere bisection. The random sphere method [9] partitions a graph by exploiting spatial information to identify separators that align with the intrinsic geometry of the vertex distribution. Instead of relying on axis-aligned or moment-based directions, this approach uses randomized geometric projections to discover low-cut partitions. The algorithm initially normalizes the vertex by translating and scaling the vertex coordinates to ensure the distribution is centered at the origin and confined within a unit-scale region. It subsequently maps the vertex coordinates onto the unit sphere via stereographic projection, thereby embedding the original Euclidean geometry into a higher-dimensional spherical manifold. To identify effective separators, the algorithm selects several approximate center points on the sphere, each obtained as the coordinate-wise median of a randomly sampled subset of vertices. Around each center point, a conformal transformation is applied to recenter the spherical embedding, thereby increasing the likelihood that sampled circles align with meaningful structures in the graph. For each transformed configuration, the algorithm samples multiple directions on the sphere, each defining a candidate circle separator. Vertices are then partitioned according to the sign of their inner product with the chosen direction vector, effectively dividing the sphere into two hemispheres.
In addition to spherical separators, the method also considers a set of random linear cuts in the original Euclidean space. Each linear cut is defined by a hyperplane orthogonal to a randomly sampled direction and positioned to bisect the vertex set at the median projection values. The final output is the spherical or linear partition that minimizes the total edge cut.
The random spherical bisection algorithm implemented in `GraphLab.jl` is described in Algorithm 3 and can be applied with:

```
GraphLab.part_randsphere(A, coords; ntrials)
```

An optional argument `ntrials` specifies the number of random directions to try.

Algorithm 3 Random sphere bisection.

Require: Graph $G = (\mathcal{V}, \mathcal{E})$, points $P_i = (x_1, \ldots, x_d)_i$
Ensure: A bisection of G into $\mathcal{V}_1$ and $\mathcal{V}_2$
1: **function** RANDOM_SPHERE_PART(graph G, points P_i)
2: Calculate the center of mass $\bar{P}$
3: Normalize $P_i \to \tilde{P}_i = (P_i - \bar{P})/\max_j |P_j - \bar{P}|$
4: Project each $\tilde{P}_i$ onto the unit sphere $\tilde{P}_i \to Z_i \in \mathcal{S}^d$
5: **for** each of m random center points **do**
6: Select a random subset of points from Z
7: Compute coordinate-wise median c
8: Apply conformal map to move c to the origin
9: **for** each random directions u **do**
10: $\mathcal{V}_1 \leftarrow i \mid \langle Z_i, u\rangle \le 0, \quad \mathcal{V}_2 \leftarrow i \mid \langle Z_i, u\rangle > 0$
11: Select $\mathcal{V}_1, \mathcal{V}_2$ with the smallest edge cut
12: **end for**
13: **end for**
14: **return** $\mathcal{V}_1, \mathcal{V}_2$
15: **end function**

3.1.4 Adaptive space-filling curves. Space-filling curves (SFCs) provide a continuous, one-dimensional traversal of multidimensional space that preserves spatial locality. In the context of partitioning, SFCs induce a linear ordering of the data points, enabling

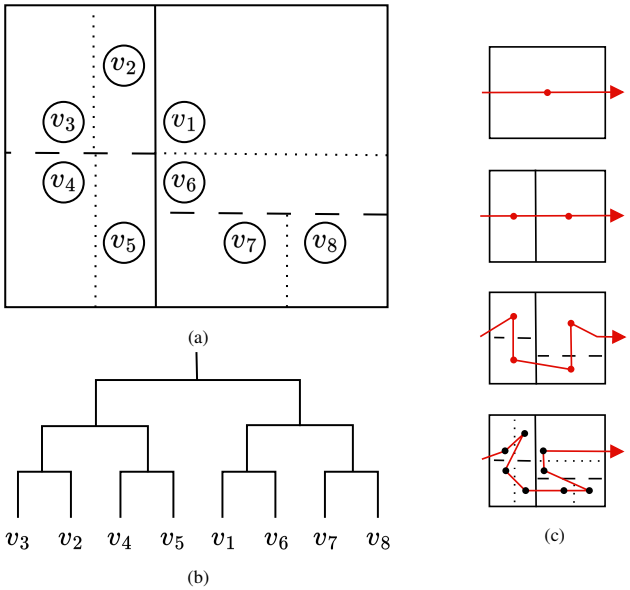


Fig. 4: The adaptive space-filling curve (SFC) partitioning process begins with the recursive spatial subdivision of the input domain (a), followed by the construction of the corresponding KD-tree (b). Then, a direction-aware recursive traversal defines a linear ordering of the leaf nodes (c). In this example, the resulting order is $v_3 \to v_2 \to v_4 \to v_5 \to v_7 \to v_8 \to v_6 \to v_1$.

recursive division into balanced and spatially coherent subregions. In this work, we implement an adaptive SFC traversal over a hierarchical spatial decomposition to generate partitions that reflect the geometric structure of the input graph [13, 14].

The algorithm, presented in Algorithm 4, performs coordinate bisection as introduced in Section *3.1.1* based on spatial distribution of the graph's vertices, using an adaptive SFC traversal. It begins by constructing a KD-tree of the vertex coordinates P, where, at each node, the splitting axis is chosen according to the direction of maximum spatial extent. The point set P is then recursively divided until each leaf contains a single point.

Once the tree is built a, an adaptive SFC traversal defines a linear order of the leaves by visiting spatial regions in a directionally consistent, locality-preserving manner. This process is governed by entry and exit directions propagated recursively. The coordinates associated with each leaf are collected in the order of traversal, yielding a one-dimensional sequence. This process is illustrated in Figure 4.

The resulting sequence is subsequently partitioned into k contiguous segments of approximately equal size, producing k spatially coherent and locality-preserving partitions. A key advantage of this method over other partitioning approaches is that, once the traversal is computed, partitioning into an arbitrary number of parts becomes a trivial post-processing step: splitting the ordered node list into contiguous segments.

Adaptive space-filling curve partitioning in `GraphLab.jl` is invoked as follows, with an optional argument `k` specifying the number of partitions:

```
GraphLab.part_adaptive_sfc(A, coords, k)
```

Algorithm 4 Adaptive Space-Filling Curve Partitioning.

Require: Graph $G = (\mathcal{V}, \mathcal{E})$, points $P_i = (x_1, x_2)_i$, number of parts k
Ensure: A partition of G into k parts
function ADAPTIVE_SFC_PARTITION(graph G, points P_i, k)
  Build spatial tree $T \leftarrow$ BUILD_TREE(P_i)
  Traverse T: $order \leftarrow$ TRAVERSE_SFC(T, L, R)
  Partition the linear $order$ into k balanced parts
  return the k partitions of $\mathcal{V}$
end function
function BUILD_TREE(points P_i)
  if $|P_i| = 1$ **then**
    return leaf node containing P_1
  else
    Compute bounding box of P_i
    Divide P_i into two subsets P^-, P^+
    Node $N_{\text{left}} \leftarrow$ BUILD_TREE(P^-)
    Node $N_{\text{right}} \leftarrow$ BUILD_TREE(P^+)
    return node with N_{left} and N_{right} as children
  end if
end function
function TRAVERSE_SFC(node N, entry, exit, accumulator a)
  if N is a leaf **then**
    exit $\leftarrow$ coord. of N
    Append N to a
    return exit
  else
    Determine child order based on SFC rules
    for each child in order **do**
      Recursively traverse child with updated entry/exit
    end for
    return a
  end if
end function

3.2 Non-geometric-based partitioning algorithms

Geometric-based partitioning algorithms are efficient techniques for partitioning meshes, particularly when spatial adjacency plays a key role in connectivity. However, these algorithms have inherent limitations. They rely on the assumption that the graph's vertices exhibit a spatial relationship, an assumption that does not hold in all contexts, such as social networks[12] or parallel processing[2]. To accommodate a broader range of applications, alternative algorithms that do not rely on geometric information have been developed. Notable examples include the Kernighan-Lin algorithm[11] and graph-growing algorithms[10], both well suited for partitioning graphs an lacking explicit spatial structure. Among these, spectral bisection is a particularly powerful technique, leveraging the eigenvalues and eigenvectors of the graph Laplacian matrix to inform partitioning decision[7]. In this subsection, we focus on the implementation and application of spectral bisection, detailing its computational properties and advantages over geometry-dependent algorithms.

3.2.1 Spectral bisection. The spectral bisection algorithm partitions a graph by leveraging the eigenvector corresponding to the second-smallest eigenvalue — commonly known as the Fiedler vector— of the graph's Laplacian matrix $\mathbf{L}$. The Laplacian matrix is defined as:

$$\mathbf{L} = \mathbf{D} - \mathbf{A}, \tag{3}$$

where $\mathbf{D}$ is the degree matrix and $\mathbf{A}$ is the adjacency matrix. The graph Laplacian $\mathbf{L}$ is a symmetric, positive semi-definite matrix, ensuring the existence of an orthogonal basis of eigenvectors $\mathbf{u}^{(i)}$ with corresponding eigenvalues $\lambda^{(i)}$. The smallest eigenvalue, $\lambda^{(1)} = 0$, and its associated eigenvector $\mathbf{u}^{(1)} = c\mathbf{1}$, where c is a constant and $\mathbf{1}$ is the all one's vector, correspond to the trivial solution that reflects the connectivity of the entire graph. The eigenvector $\mathbf{u}^{(2)}$ associated with the second-smallest eigenvalue $\lambda^{(2)}$, known as the Fiedler vector[7], captures intrinsic connectivity pattern of the graph, as illustrated in Figure 5.

Each node v_i is associated with the corresponding entry $\mathbf{u}_i^{(2)}$ of the Fiedler vector. The partition is performed by thresholding these values:

—A threshold at zero yields two roughly balanced subsets while minimizing the edge cut.

—A threshold at $\mathbf{u}^{(2)}$ results in two strictly equal-sized partitions.

The complete spectral bisection algorithm is outlined in Algorithm 5, detailing its key computational steps.

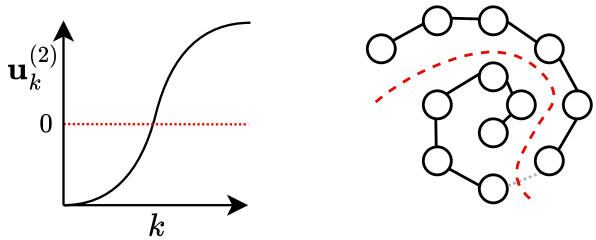


Fig. 5: Plot of the values of $\mathbf{u}^{(2)}$, where each $\mathbf{u}_i^{(2)}$ corresponds to vertex v_i in the original sequence, and where $\mathbf{u}_k^{(2)}$ corresponds to v_k in the reordered sequence, such that $\mathbf{u}_1^{(2)} < \mathbf{u}_2^{(2)} < \cdots < \mathbf{u}_n^{(2)}$ (left). The graph partitions corresponding to this ordering (right), with the threshold $\epsilon = 0$ defining the cut (right).

Algorithm 5 Spectral bisection.

Require: Graph $G = (\mathcal{V}, \mathcal{E})$
Ensure: A bisection of G into $\mathcal{V}_1$ and $\mathcal{V}_2$
1: **function** SPECTRAL_PART(graph G)
2: Form the Laplacian matrix $\mathbf{L}$
3: Compute the 2nd smallest eigenval. $\lambda^{(2)}$ and eigenvec. $\mathbf{u}^{(2)}$
4: Set 0 or the median of $\mathbf{u}^{(2)}$ as threshold ϵ
5: Set $\mathcal{V}_1 := \{v_i \in \mathcal{V} \mid u_i < \epsilon\}$, $\mathcal{V}_2 := \{v_i \in \mathcal{V} \mid u_i \geq \epsilon\}$
6: **return** $\mathcal{V}_1, \mathcal{V}_2$
7: **end function**

Spectral bisection method implemented in `GraphLab.jl` can be executed with:

```
GraphLab.part_spectral(A)
```

It requires only the adjacency matrix `A` as input and returns a vector assigning each node to partition 1 or 2.

3.3 Hybrid partitioning algorithms

Building on the random sphere bisection and the spectral partitioning described in *3.1.3* and *3.2.1*, hybrid bisection extends spectral methods with a randomized geometric layer to enhance partitioning quality, particularly in graphs with an underlying spatial structure.
The method begins by computing a spectral embedding of the graph, in which each vertex is mapped to a point in $\mathbb{R}^d$ using the first d nontrivial eigenvectors of the Laplacian matrix. This embedding encodes the global connectivity structure of the graph in a continuous geometric space, often revealing natural separations aligned with sparse cuts.
The embedded point cloud is partitioned with the random sphere method. This process effectively performs a geometric search over separators, guided by the spectral structure of the graph.
The hybrid method leverages the algebraic properties of spectral embeddings alongside the geometric adaptability of random sphere cuts to generate balances and spatially localized partitions.
Geometric spectral partitioning in `GraphLab.jl` can be executed with:

```
GraphLab.part_geospectral(A; ev=d)
```

In addition to the adjacency matrix `A`, it optionally accepts `ev` $= d$ the number of nontrivial Laplacian eigenvectors to use for embedding (default: 2).

3.4 Recursive bisection and nested dissection

Recursive bisection and nested dissection are techniques that rely on recursively partitioning a graph into smaller subgraphs. While recursive bisection is primarily used to generate multiple balanced partitions, nested dissection applies a recursive strategy to reduce fill-in during sparse matrix factorization. This section outlines both methods and their respective algorithmic formulations.

3.4.1 Recursive bisection. A straightforward and effective strategy for partitioning a graph into $p = 2^q$ parts, where q is a positive integer, is recursive bisection, as presented in Algorithm 6. This algorithm iteratively applies graph bisection, progressively subdividing the graph into smaller sub-graphs. In `GraphLab.jl`, recursive bisection can be used with any of the bisection algorithms presented in Sections 3.1 and 3.2. The algorithm is built around a recursive function, `Recursion`, which takes as inputs:

—C', the current sub-graph to be partitioned,

—p', the number of partitions into which C' will be further divided, and

—idx, an integer tracking the position of the first part of C' in the final partitioning results.

At each recursive step, the sub-graph C' is bisected into two balanced parts. The process continues until the desired number of partitions, $p = 2^q$, is obtained. This recursive strategy results in a structured, hierarchical decomposition of the graph, making it particularly well suited for parallel computing applications, especially in domains like finite element and finite difference methods[16].
`GraphLab.jl` provides a recursive interface for all partitioning algorithms described in Section 3, which can be invoked with:

```
GraphLab.recursive_bisection(method, k,
    A, coords)
```

Algorithm 6 Recursive bisection.

Require: Graph $G = (\mathcal{V}, \mathcal{E})$
Ensure: A p-way partition of G
$G_p = \{C_1, \ldots, C_p\}$
$p = 2^l$
function REC_BISECTION(graph G, number of parts p)
  function RECURSION(C', p', idx)
    if p' is even **then**
      $p' \leftarrow \frac{p'}{2}$
      $(C'_1, C'_2) \leftarrow$BISECTION(C')
      RECURSION(C'_1, p', idx)
      RECURSION(C'_2, p', idx + p')
    else
      $C_{\text{idx}} \leftarrow C'$
    end if
  end function
  RECURSION(C, p, 1)
  return G_p
end function

Here, `method` is any partitioning function available in the package, `k` is the desired number of partitions (if not a power of two, it is automatically rounded up to the nearest power of two), and `A` and `coords` are the graph's adjacency matrix and node coordinates. The function returns a vector assigning each node a label from 1 to `k`, indicating its partition membership.

3.4.2 Nested dissection. The nested dissection ordering algorithm is a multilevel heuristic introduced to minimize fill-in, i.e., the creation of nonzero entries during sparse matrix factorizations[8]. It recursively partitions a graph G through the identification of balanced vertex separators, which are removed to decompose G into disconnected components. The same procedure is then applied recursively to each subgraph. Unlike standard recursive bisection, as described in Section *3.4.1*, which directly bisects the graph into two parts, nested dissection introduces an intermediate step: the explicit computation of a separator whose removal divides the problem into independent subproblems. To compute the separator, border vertices are first identified between the two subdomains resulting from the initial bisection. These vertices induce a bipartite graph, where edges represent adjacency across the partition boundary. A maximum matching is then computed on this bipartite graph; this matching corresponds to a minimum vertex cover[18, 17], providing an efficient approximation of a small separator. The selected separator vertices are removed, and the nested dissection proceeds recursively on the resulting components. The final ordering π places all separator vertices after the recursively ordered interior vertices, yielding a global vertex ordering that preserves sparsity patterns. The overall process is outlined in Algorithm 7.
This strategy is widely used in the symbolic factorization phase of sparse direct solvers, where it facilitates the construction of efficient elimination trees and reduces both fill-in and memory overhead during numerical factorization. Nested dissection can be implemented using partitioning methods presented in Section 3.1 and Section 3.2 for separator computation, making it a flexible tool in both partitioning and numerical linear algebra contexts.
The nested dissection provided by `GraphLab.jl` can be called using:

```
GraphLab.nested_dissection(A, method;
    coords, minsep=5)
```

Algorithm 7 Nested dissection ordering.

Require: Adj. matrix A, partitioning METHOD, minimum separator size minsep
Ensure: Permutation vector π s.t. $A[\pi, \pi]$ has reduced fill-in
function NESTED_DIS(A, METHOD, coords (opt.), minsep)
  Identitfy connected components of A
  Initialize permutation vector π
  for each component C **do**
    if $|C| \leq$ minsep **then**
      Apply minimum degree ordering on $A[C, C]$
    else
      Bisect $C \to C_1, C_2$ via METHOD
      Identify separator nodes between C_1 and C_2
      Recursively compute orderings:
        $\pi_1 \leftarrow$ NESTED_DIS($A[C_1, C_1]$)
        $\pi_2 \leftarrow$ NESTED_DIS($A[C_2, C_2]$)
      Combine $\pi_C \leftarrow [\pi_1, \pi_2, \text{separator}]$
    end if
    Insert π_C into global permutation π
  end for
  return π
end function

The inputs are the adjacency matrix `A`, any partitioning `method` from `GraphLab.jl`, and the node coordinates `coords` if required by the chosen method. An optional argument `minsep` specifies the minimum separator size (default: 5). The output is a permutation vector representing the nested dissection ordering.

4. Framework and tools for graph bisection

While a variety of algorithms exist for performing graph bisection, as discussed in Section 3, an integrated framework is essential to streamline the entire workflow — from graph creation and algorithm execution to benchmarking and visualization. Existing tools often address only specific aspects of this process, but lack cohesive support for tasks like generating graphs, managing input/output, benchmarking performance, and visually interpreting results. We propose a comprehensive framework for graph bisection, designed to unify these components into a single, structured workflow. The framework consists of the following core modules:

(1) **Graph creation**: Generating and managing graphs suitable for the application of various partitioning algorithms.
(2) **Graph bisection**: Implementing the bisection methods detailed in Section 3.
(3) **Benchmarking**: Measuring and comparing algorithm performance according to graph-cut criteria.
(4) **Visualization**: Providing visual representations of graphs and their partitions to facilitate analysis and comparison.
(5) **Integration**: Enabling interoperability with existing graph partitioning tools and libraries.

4.1 Graph creation

Graphs used in the framework can either be syntheticaly generated or loaded from external files. Synthetic graphs are typically generated as $n \times m$ grids with a rotation of θ radians. Alternatively, users can upload a `mat` file with an adjacency matrix and optional coordinate, or a `csv` file with coordinates only, from which the adjacency matrix is constructed using k-nearest neighbors. The file parsing

and data loading are handled by the external packages `MAT.jl`[1] and `DelimitedFiles.jl`[2].

4.2 Benchmarking

The framework includes two example scripts, provided in the `examples` directory of the `GraphLab.jl` package, to benchmark the implemented graph bisection strategies. The edge cut is defined as the number of edges crossing between partitions, while the balance ratio measures how evenly the graph is divided.

(1) `ex1.jl`: Benchmarks multiple bisection methods across a set of mesh inputs. For each method and mesh, it evaluates the edge cut and partition balance. Since all resulting balance ratios are very close to one, we report the achieve edge cut values in Table 1.

(2) `ex2.jl`: Benchmarks recursive bisection. The graph is recursively partitioned into $p = 8$ and $p = 16$ subdomains using a given base bisection method. Edge cut and balance are recorded for each case.

4.3 Visualization

To generate visual representations of graph partitions, this steps requires the adjacency matrix $\mathbf{A}$, the vertex coordinates, and the corresponding partition information. The framework integrates multiple Julia packages to ensure clear visualization:

—`SGtSNEpi.jl`[3]: A scalable tools for embedding and plotting large graphs, addressing challenges posed by visualization libraries that struggle with size constraints.

—`Graphs.jl`[4]: Provides essential graph operations and data structures.

—`CairoMakie.jl`[5]: Enables the creation customizable plots suitable for publication.

—`Colors.jl`[6]: Manages color palettes for visually distinguishing partitions.

Visualizations of the results produced by the scripts introduced in Section 4.2 are presented in Figure 6 and Figure 7.

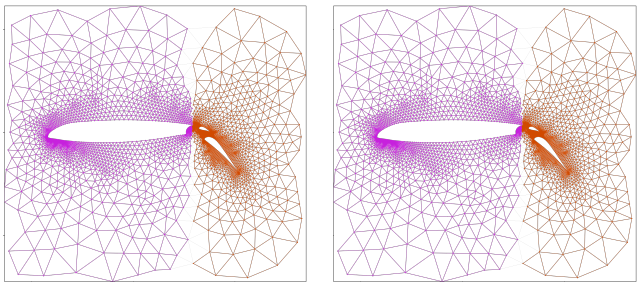
5. Installation and demonstration

To install the package from GitHub[7] and integrate it into the working environment, the following steps are required:

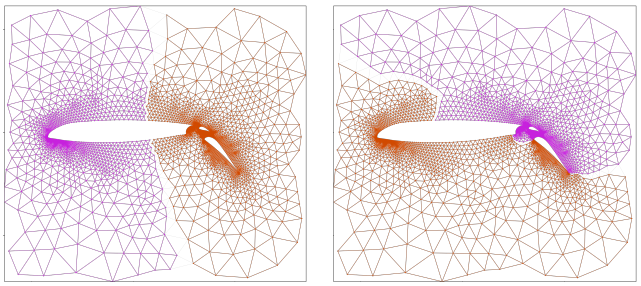
(1) Add `GraphLab.jl` to the project using the Julia command:

```
using Pkg
Pkg.add(url="https://github.com/lechekhabm/
    ↪ GraphLab.jl")
```

[1] `https://github.com/JuliaIO/MAT.jl`
[2] `https://docs.julialang.org/en/v1/stdlib/DelimitedFiles/`
[3] `https://github.com/fcdimitr/SGtSNEpi.jl`
[4] `https://juliagraphs.org/Graphs.jl/`
[5] `https://docs.makie.org/stable/explanations/backends/cairomakie.html`
[6] `https://juliagraphics.github.io/Colors.jl/`
[7] `https://github.com/lechekhabm/GraphLab.jl`

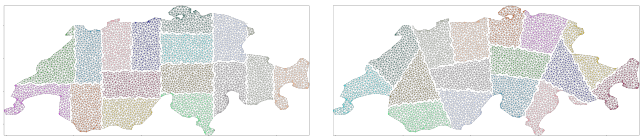


(a) Coordinate bisection (b) Inertial bisection

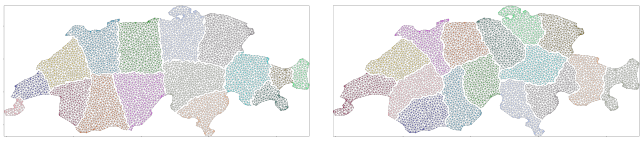


(c) Spectral bisection (d) Bisection using `METIS`

Fig. 6: Visualization of four graph bisection methods applied to the `airfoil1` mesh (4253 nodes and 12289 edges), illustrating differences in partitioning structure and edge cuts.



(a) Recursive coordinate bisection (b) Recursive inertial bisection



(c) Recursive spectral bisection (d) Recursive `METIS` bisection

Fig. 7: Comparison of four graph recursive bisection methods applied to the `Swiss_graph` (4468 nodes and 15230 edges), illustrating differences in recursive partitioning results.

(2) As a basic example for graph partitioning, the adjacency matrix $\mathbf{A}$ and vertex coordinates are first constructed from the input data. Here, we generate a synthetic 10×50 rectangular grid graph rotated by an angle of $\pi/3$ radians. Spectral bisection is then applied to compute the partition p, followed by visualization and export of the partitioned graph as an image. The process is executed with the following commands:

```
using GraphLab
A, coords = GraphLab.grid_graph(10, 50, π/3)
p = GraphLab.part_spectral(A)
GraphLab.draw_graph(A, coords, p, file_name="
    ↪ test.png")
```

Table 1. : Edge cuts for each method and mesh.

Mesh	coordinate	inertial	randsphere	adaptive sfc	spectral	geospectral	METIS
`3elt`	172	209	94	224	117	117	90
`airfoil1`	94	94	93	98	132	132	73
`barth4`	206	194	130	208	127	127	100
`crack`	323	377	275	353	233	233	200
`mesh1e1`	18	19	18	18	18	18	17
`mesh2e1`	37	47	36	40	35	35	34
`mesh3e1`	17	32	18	21	30	20	18
`mesh3em5`	17	32	18	21	18	20	18
`netz4504_dual`	25	30	23	25	23	23	20
`stufe`	16	16	16	16	16	16	17
`ukerbe1`	27	28	34	34	28	28	30

Further details on the package and its functionalities can be found in the online documentation[8].

6. Conclusion and future works

In this work, we have presented `GraphLab.jl`, a comprehensive and extensible framework for graph partitioning, designed to support both research and education in graph theory and partitioning problems. By integrating foundational partitioning techniques — such as coordinate, inertial, and spectral bisection, as well as random spheres, space-filling curves, and nested dissection — together utilities for visualization, benchmarking, and quality assessment, the framework provides a structured and interactive environment for analyzing and comparing graph partitioning strategies.
A key strength of `GraphLab.jl` lies in its modular design and extensibility. Users can experiment with a diverse range of partitioning algorithms, leveraging both native implementations and external software, all within a unified and reproducible setting.
Ongoing developments aim to broaden the framework's capabilities, with planned extensions to partitioning techniques, evaluation metrics, and overall performance.

7. Acknowledgment

The authors gratefully acknowledge the scientific support and HPC resources provided by the Erlangen National High Performance Computing Center (NHR@FAU) of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) under the NHR project j101df. NHR funding is provided by federal and Bavarian state authorities. NHR@FAU hardware is partially funded by the German Research Foundation (DFG) – 440719683 We also would like to acknowledge the financial support of the joint DFG (ID 470857344) and SNSF (ID 200021L_204817) project entitled *Numerical Algorithms, Frameworks, and Scalable Technologies for Extreme-Scale Computing*, and the computing support by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID u3-31045.

8. References

[1] Charles-Edmond Bichot. *General Introduction to Graph Partitioning*, chapter 1, pages 1–25. John Wiley & Sons, Ltd, 2013.

[2] Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[3] Aydin Buluc, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning, 2015.

[4] Ümit Çatalyürek, Karen Devine, Marcelo Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More recent advances in (hyper)graph partitioning. *ACM Comput. Surv.*, 55(12), March 2023.

[5] Ulrich Elsner. Graph partitioning-a survey. 1997.

[6] Charbel Farhat and Marc Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *International Journal for Numerical Methods in Engineering*, 36(5):745–764, 1993.

[7] Miroslav Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(4):619–633, 1975.

[8] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[9] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[10] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[11] Brian Wilson Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[12] M. E. J. Newman. Community detection and graph partitioning. *EPL (Europhysics Letters)*, 103(2):28003, July 2013.

[13] Aparna Sasidharan, John M. Dennis, and Marc Snir. A general space-filling curve algorithm for partitioning 2d meshes. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 875–879, 2015.

[14] Aparna Sasidharan and Marc Snir. Space-filling curves for partitioning adaptively refined meshes. *Mathematics and Computer Science*, 2015.

[8] `https://lechekhabm.github.io/GraphLab.jl/dev/`

[15] Horst D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2):135–148, 1991. Parallel Methods on Large-scale Structural Analysis and Physics Applications.

[16] Horst D. Simon and Shang-Hua Teng. How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, 1997.

[17] James Andrew Storer. *An introduction to data structures and algorithms*. Springer Science & Business Media, 2012.

[18] Jacob Turner. Mapping matchings to minimum vertex covers: König's theorem revisited, 2020.