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
[2]Seminar for Applied Mathematics, ETH Zürich

ABSTRACT

We design and implement `GraphLab.jl`, a `Julia` package facilitating the study, experimentation, and research of graph partitioning. `GraphLab.jl` explores the principles and trade-offs of partitioning algorithms. It offers a 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.

Keywords

Julia, Graph algorithms, Graph partitioning, Recursive methods, Numerical computing

1. Introduction

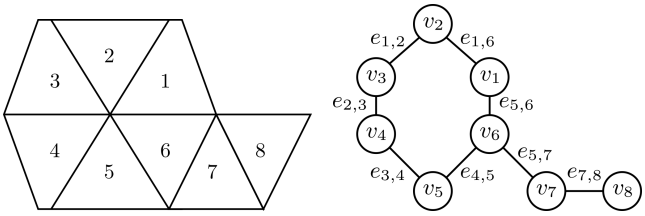
Graph partitioning is a fundamental problem with wide-ranging applications in computational biology, high-performance computing, and distributed systems, among many other domains [5]. Partitioning large graphs into loosely connected subsets of roughly equal size promotes parallel execution, reduces communication overhead, and provides insights into the structure of complex networks [6].

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. Similar to existing toolboxes such as `meshpart` [15] in `MATLAB`, and `CDLIB` [27] in `Python`, which focuses on community detection, `GraphLab.jl` aims to offer a framework for graph partitioning in `Julia`. A diverse set of partitioning algorithms are implemented in the introduced package, including coordinate [31], inertial [12], and spectral bisection [13, 31], random spheres [16], space-filling curves [29], and nested dissection [14]. These methods can be applied recursively for hierarchical partitioning or fill-in reducing strategies. `GraphLab.jl` also provides routines for generating adjacency matrices based on coordinate systems, computing partition quality metrics, benchmarking problems, and visualizing partitioned graphs.

The paper is structured as follows. Section 2 provides a brief background on graph partitioning, followed by an overview of the fundamental implemented partitioning algorithms in Section 3. Section 4 introduces our framework, detailing its capabilities in graph creation, partitioning, benchmarking and visualization. Installation and usage demonstrations are presented 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 1. If data exchange occurs only between adjacent cells, the mesh can be represented as a dependency graph. To partition the mesh into two domains suitable for parallel processing, the objective is to divide it into two parts of roughly equal size, while minimizing the number of edges connecting them. This corresponds to partitioning the original graph into two complementary subgraphs, that have an almost equal number of vertices and a minimum number of interconnecting edges between them.



(a) (b)

Fig. 1: Mesh example consisting of 8 cells (a) and the corresponding dependency graph (b).

Finding an optimal solution is NP-hard for this bisection problem, making the exact computation intractable for large instances [5]. In the following, we formalize the graph partitioning problem and introduce bisection algorithms. Here, bisection specifically refers to partitioning the graph into two subgraphs. A general partitioning into $p = 2^l$ sub-graphs, where l denotes the number of recursive bisection levels, can then be obtained recursively by applying a bisection method 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 captures the connectivity among its vertices, with the entry $\mathbf{A}_{ij}$ 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 non-negative real-valued number indicating the strength 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 $a_{ij} = a_{ji} = 1$ 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 adjacent 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 [2] for a detailed overview of commonly used matrices and objectives functions in graph partitioning.

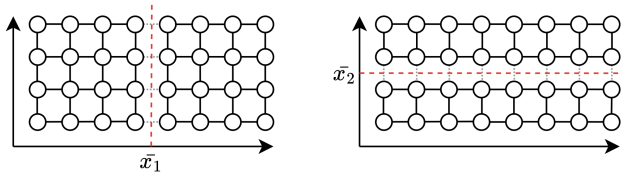
3. Overview of implemented partitioning algorithms

This section provides an overview of the partitioning algorithms implemented in the framework, detailing their underlying principles. 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 [16]. 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 [5]. 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 a two-dimensional example illustrated in Figure 2. The complexity of coordinate bisection is $O(d(n+m))$ when using linear-time median selection or $O(d(n \log n + m))$ with sorting-based medians [7], where n is the number of vertices, and m the number of edges.
The coordinate bisection method in `GraphLab.jl` can be invoked using the following command:



(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 $\mathcal{V}_2$
function COORDINATE_PART(graph G, points P_i)
  Initialize $c_{\min} \leftarrow \infty$, $j^* \leftarrow 1$
  for each axis x_j, $j = 1, \ldots, d$ **do**
    Compute the median $\bar{x}_j$
    Compute the edge cut c_j for the bisection at $\bar{x}_j$
    if $c_j < c_{\min}$ **then**
      $c_{\min} \leftarrow c_j$
      $j^* \leftarrow j$
    end if
  end for
  Partition $\mathcal{V}$ into $\mathcal{V}_1$ and $\mathcal{V}_2$ via $\bar{x}_{j^*}$ bisection
  return $\mathcal{V}_1, \mathcal{V}_2$
end function

```
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.

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. Physically, this direction corresponds to the axis of minimal rotational inertia [10], ensuring 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 $l(\lambda) = \{\bar{P} + \lambda\mathbf{u} \mid \lambda \in \mathbb{R}\}$.
To determine the optimal orientation of the bisecting hyperplane, the unit direction vector $\mathbf{u}$ is chosen to minimize the sum of the

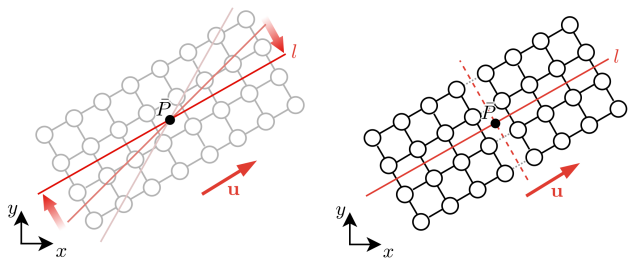


Fig. 3: Illustration of the inertial bisection in 2D: dividing the graph along the line orthogonal to the direction $\mathbf{u}$ through the center of mass $\bar{P}$ that minimizes the sum of squared distances.

squared distances from the vertices to the line. In the case of a two-dimensional graph embedding, this reads:

$$\begin{aligned}\sum_{i=1}^{n} d_i^2 &= \sum_{i=1}^{n}(x_i-\bar{x})^2+(y_i-\bar{y})^2 \\ &\quad -(u_1(x_i-\bar{x})+u_2(y_i-\bar{y}))^2 \\ &= (1-u_1^2)\sum_{i=1}^{n}(x_i-\bar{x})^2+(1-u_2^2)\sum_{i=1}^{n}(y_i-\bar{y})^2 \\ &\quad +2u_1u_2\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y}) \\ &= (1-u_1^2)S_{xx}+(1-u_2^2)S_{yy}+2u_1u_2S_{xy} \\ &= \mathbf{u}^T\begin{pmatrix}S_{xx} & S_{xy}\\ S_{xy} & S_{yy}\end{pmatrix}\mathbf{u}=\mathbf{u}^T\mathbf{M}\mathbf{u} \end{aligned} \tag{3}$$

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 $\mathbf{M}$ [10]. 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. The inertial bisection runs in time $O(nd+m+d^3)$, which simplifies to $O(n+m)$ in fixed spatial dimension.

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)
```

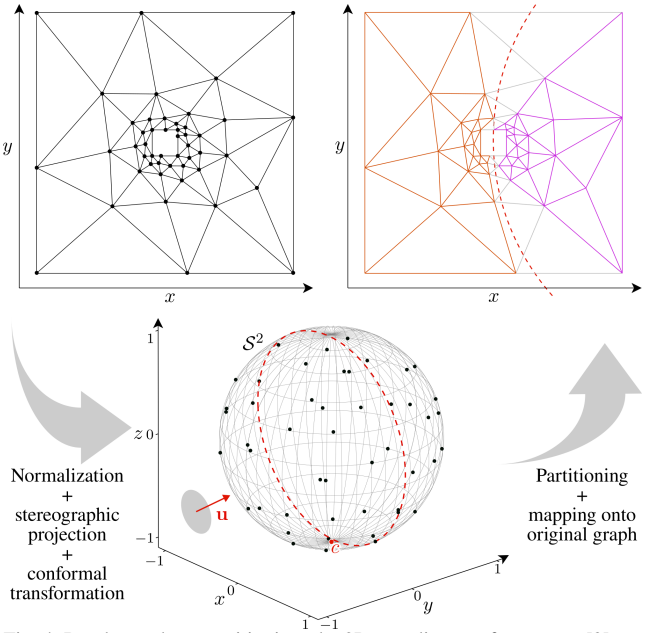


Fig. 4: Random sphere partitioning: the 2D coordinates of `mesh1e1` [9] are normalized and stereographically lifted to the sphere $\mathcal{S}^2 \subset \mathbb{R}^3$, where a centerpoint c is computed and mapped to the origin via a conformal transformation. A direction $\mathbf{u}$ defining a circle separator (shown in red) is then selected, partitioning the spherical embedding. The resulting partition is projected onto the original layout.

3.1.3 Random sphere bisection. The random sphere method [16] partitions a graph by exploiting spatial information to identify separators aligned with the intrinsic geometry of the vertex distribution. Unlike axis-aligned or inertia-based approaches, it employs randomized geometric projections to discover low-cut partitions.
Given vertex coordinates $P_i=(x_1,\ldots,x_d)_i$, the algorithm first computes the center of mass $\bar{P}$ and normalizes the coordinates as

$$\tilde{P}_i=\frac{P_i-\bar{P}}{\max_j|P_j-\bar{P}|}, \tag{4}$$

so that the distribution is centered at the origin and confined within a unit-scale region. Each normalized point $\tilde{P}_i$ is then mapped to the unit sphere, $Z_i\in\mathcal{S}^d\subset\mathbb{R}^{d+1}$, via stereographic projection.
To identify separators, the algorithm selects s random center points on the sphere. Each center c is computed as the coordinate-wise median of a randomly sampled subset of vertices and then moved to the origin through a conformal transformation of the sphere. For each transformed configuration, several random unit directions u are sampled, and a candidate spherical cut is generated as

$$\mathcal{V}_1=\{i\mid\langle Z_i,u\rangle\le 0\},\quad \mathcal{V}_2=\{i\mid\langle Z_i,u\rangle>0\}, \tag{5}$$

The corresponding edge cut is evaluated, and the partition with the smallest cut is retained.
In addition to spherical separators, the algorithm also considers random linear cuts in the original Euclidean space, defined by hyperplanes orthogonal to random directions and positioned at the median projected vertex coordinates. The final output is the spherical or linear bisection minimizing the total edge cut.
The random spherical bisection runs in $O(sr(nd+m))$, where s is the number of random centers and r the number of random

directions per center. For fixed d, s, and r, this reduces to $O(n+m)$. The algorithm 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. A geometric illustration of the random sphere bisection procedure, applied to 2-dimensional input coordinates and visualized on the sphere $\mathcal{S}^2 \subset \mathbb{R}^3$, is provided in Figure 4.

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 $\mathcal{V}_2$

function RANDOM_SPHERE_PART(graph G, points P_i)
  Calculate the center of mass $\bar{P}$
  Normalize $P_i \to \tilde{P}_i = (P_i - \bar{P}) / \max_j |P_j - \bar{P}|$
  Project each $\tilde{P}_i$ onto the unit sphere $\tilde{P}_i \to Z_i \in \mathcal{S}^d$
  for each of s random center points **do**
    Select a random subset of points from Z
    Compute coordinate-wise median c
    Apply conformal map to move c to the origin
    for each random directions u **do**
      $\mathcal{V}_1 \leftarrow i \mid \langle Z_i, u \rangle \leq 0, \quad \mathcal{V}_2 \leftarrow i \mid \langle Z_i, u \rangle > 0$
      Select $\mathcal{V}_1, \mathcal{V}_2$ with the smallest edge cut
    end for
  end for
  return $\mathcal{V}_1, \mathcal{V}_2$
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 recursive division into balanced and spatially coherent subregions. We implement an adaptive SFC traversal over a hierarchical spatial decomposition to generate partitions that reflect the geometric structure of the input graph [29, 30].

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 k-dimensional tree (k-d tree) of the vertex coordinates P, a binary space-partitioning structure that recursively subdivides the domain along axis-aligned hyperplanes. At each node of the k-d tree, the splitting axis is chosen as the direction of maximum spatial extent, and the point set P is recursively divided until each leaf contains a single point.

Once the tree is built, an SFC traversal induces a linear order of the leaves by visiting spatial regions in a directionally consistent, locality-preserving manner. At each recursive step, the traversal maintains entry and exit directions that specify how the curve enters and leaves a region, ensuring a continuous path between adjacent subregions. The coordinates associated with each leaf are collected in traversal order, producing a one-dimensional sequence of vertices. This process is illustrated in Figure 5.

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. Construction of the adaptive spatial tree requires $O(n \log n)$ time, and the subsequent SFC traversal and linear partition each cost $O(n)$, giving an overall complexity of $O(n \log n)$.

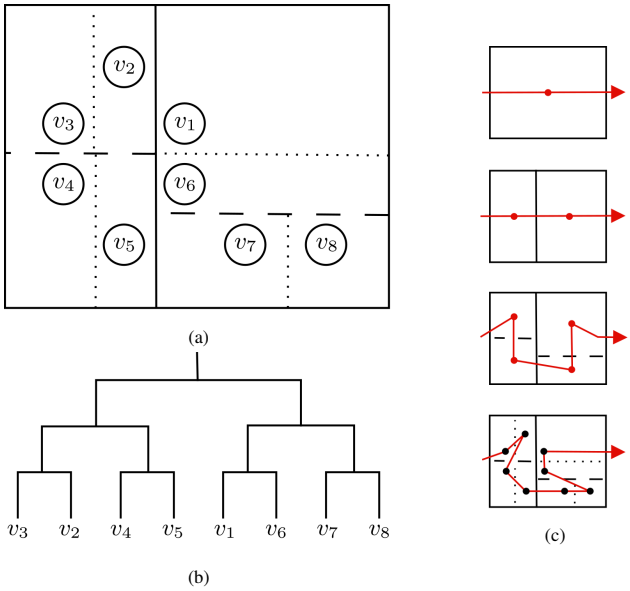


Fig. 5: The adaptive space-filling curve (SFC) partitioning process begins with the recursive spatial subdivision of the input domain (a), followed by the construction of the corresponding k-d tree (b). Then, a direction-aware recursive traversal defines a linear ordering of the leaf nodes (c). In this example, the resulting order is $v_3 \to v_2 \to v_4 \to v_5 \to v_7 \to v_8 \to v_6 \to v_1$.

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)
```

3.2 Non-geometric-based partitioning algorithms

Geometric-based partitioning algorithms rely on the premise that the graph's vertices exhibit a spatial relationship, a condition that does not hold in all contexts, such as social [24] or biological [1] networks.

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 [19] and graph-growing algorithms [18], both well suited for partitioning graphs lacking explicit spatial structure. Among these, spectral bisection leverages the eigenvalues and eigenvectors of the graph Laplacian matrix to inform partitioning decision [13]. 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 combinatorial graph

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

Laplacian matrix is defined as $\mathbf{L} = \mathbf{D} - \mathbf{A}$, where $\mathbf{D}$ is the degree and $\mathbf{A}$ is the adjacency matrix. The graph Laplacian $\mathbf{L}$ is a symmetric, positive semi-definite matrix, which admits an orthonormal basis of eigenvectors $\mathbf{u}^{(i)}$ with corresponding nonnegative eigenvalues $\lambda^{(i)} \geq 0$. The smallest eigenvalue is $\lambda^{(1)} = 0$, and its associated eigenvector $\mathbf{u}^{(1)} = c\mathbf{1}$, where c is a constant and $\mathbf{1}$ the all one's vector corresponding to the trivial solution, in which all vertices of a connected graph belong to a single connected component. The eigenvector $\mathbf{u}^{(2)}$ associated with the second-smallest eigenvalue $\lambda^{(2)}$, known as the Fiedler vector [13], provides a one-dimensional embedding of the graph that reflects its connectivity structure, as illustrated in Figure 6. Each vertex v_i is assigned the scalar value $\mathbf{u}_i^{(2)}$, and vertices with similar values tend to be more tightly connected within the graph. A bisection is obtained by thresholding $\mathbf{u}^{(2)}$:

$$\mathcal{V}_1 = \{v_i \mid \mathbf{u}_i^{(2)} \leq \epsilon\}, \quad \mathcal{V}_2 = \{v_i \mid \mathbf{u}_i^{(2)} > \epsilon\}, \tag{6}$$

where $\epsilon = 0$ yields a cut that approximately minimizes the edge weight between subsets, and $\epsilon = \text{median}\left(\mathbf{u}_1^{(2)}, \ldots, \mathbf{u}_n^{(2)}\right)$ enforces a balanced partition. Spectral bisection is dominated by the Fiedler vector computation. Forming the Laplacian from a given graph costs $O(n + m)$ [25], while ARPACK's Lanczos method [22, 20] extracts the second eigenpair in $O(mt)$ time [17], where t is the number of iterations. Thus the total complexity is $O(mt)$, with all other operations linear. The spectral algorithm, outlined in Algorithm 5, can be executed with:

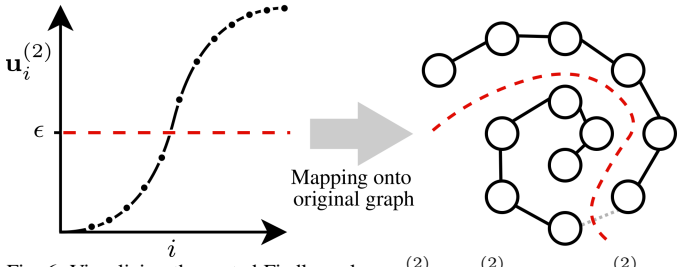


Fig. 6: Visualizing the sorted Fiedler values $\mathbf{u}_1^{(2)} < \mathbf{u}_2^{(2)} < \cdots < \mathbf{u}_n^{(2)}$, where each entry $\mathbf{u}_i^{(2)}$ corresponds to the vertex assigned to position i in the ordering (left), and their mapping to the resulting graph partition (right) obtained by thresholding $\mathbf{u}^{(2)}$ at a value ϵ, set to 0 or the median.

Algorithm 5 Spectral bisection.

Require: Graph $G = (\mathcal{V}, \mathcal{E})$
Ensure: A bisection of G into $\mathcal{V}_1$ and $\mathcal{V}_2$

function SPECTRAL_PART(graph G)
  Form the Laplacian matrix $\mathbf{L}$
  Compute the 2nd smallest eigenval. $\lambda^{(2)}$ and eigenvec. $\mathbf{u}^{(2)}$
  Set 0 or the median of $\mathbf{u}^{(2)}$ as threshold ϵ
  Set $\mathcal{V}_1 := \{v_i \in \mathcal{V} \mid u_i < \epsilon\}$, $\mathcal{V}_2 := \{v_i \in \mathcal{V} \mid u_i \geq \epsilon\}$
  return $\mathcal{V}_1, \mathcal{V}_2$
end function

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

Its sole input is the adjacency matrix `A` and its output a vector assigning each node to partition 1 or 2.

3.3 Hybrid partitioning algorithms

Combining the random sphere bisection and the spectral partitioning described in *3.1.3* and *3.2.1*, hybrid bisection extends spectral methods with a geometric layer to enhance partitioning quality, particularly in graphs with an underlying spatial structure. It begins by computing a spectral embedding of the graph, where each vertex v_i is mapped to a point

$$\mathbf{z}_i = \left(\mathbf{u}_i^{(2)}, \ldots, \mathbf{u}_i^{(k+1)}\right) \in \mathbb{R}^k, \tag{7}$$

using the first k nontrivial eigenvectors of the Laplacian matrix. This embedding encodes the global connectivity structure of the graph in a low-dimensional space, where geometric separations of the embedded points with the random sphere method lead to sparse graph cuts. This process performs a geometric search over separators, guided by the spectral structure of the graph. The figure Figure 7 illustrates the case with $k = 2$, where the embedding uses the first two non-trivial Laplacian eigenvectors. The hybrid method leverages the algebraic properties of spectral embeddings and effectiveness of random sphere cuts to generate balanced and spatially localized partitions. It's overall complexity is dominated by the spectral stage, $O(mt_k)$ for sparse graphs, where t_k is the number of Lanczos iterations required to compute k nontrivial eigenvectors.
Geometric spectral partitioning in `GraphLab.jl` can be executed with:

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

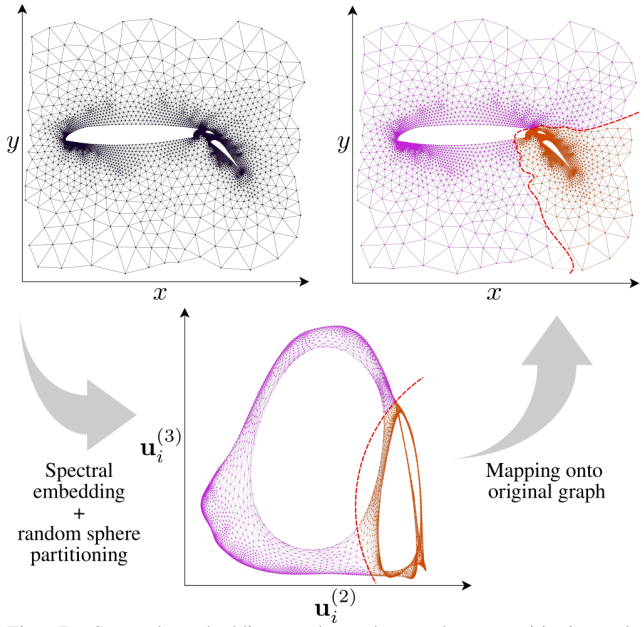


Fig. 7: Spectral embedding and random sphere partitioning: the `airfoil1` [9] graph is first embedded into a 2-dimensional spectral space using the eigenvectors associated with the second and third smallest eigenvalues of the graph Laplacian. A separator (shown in red) is then selected using the randomized sphere method and mapped back onto original graph.

In addition to the adjacency matrix `A`, it optionally accepts `ev` $= d$, i.e., 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 splitting 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 [14]. 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 subgraphs. 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 subgraph 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 subgraph C' is bisected into two approximately balanced parts. The process continues until the desired number of partitions, $p = 2^q$, is obtained. The total cost is $O(T_{\text{bisect}}(n, m) \log p)$, where $T_{\text{bisect}}(n, m)$ is the cost of a single bisection method. This recursive strategy results in a structured, hierarchical decomposition of the graph, making it particularly well suited for load balancing and for reducing communication overhead in parallel finite element and finite difference implementations [32].

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

`GraphLab.jl` provides a recursive interface for all partitioning algorithms described in Section 3, which can be invoked with:

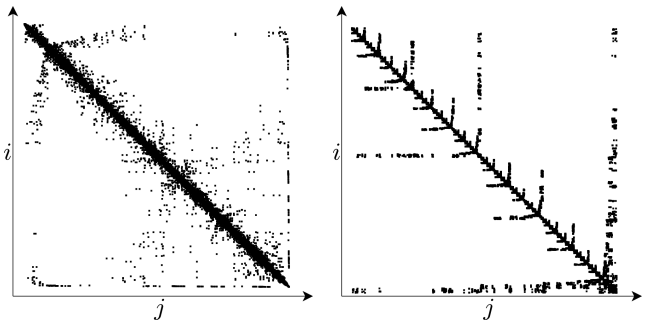
```
GraphLab.recursive_bisection(method, k,
    A, coords)
```

Here, `method` is any partitioning function available in the package, `k` is the desired number of partitions (automatically rounded up to the nearest power of two if needed), `A` is the graph's adjacency matrix, and `coords` is an optional argument (default: `nothing`) used only for coordinate-based methods. 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 designed to minimize fill-in, i.e., the creation of nonzero entries during sparse matrix factorizations [14]. 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, which directly bisects the graph into two parts, nested dissection introduces an intermediate step: the explicit computation of a node 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, in which edges represent adjacency across the partition boundary. A maximum matching is then computed on this bipartite graph and a minimum vertex cover can be derived from it [4, 33], 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 reveals a hierarchical block structure in the reordered matrix. The complexity of nested dissection is $O(T_{\text{bisect}}(n, m) \log n)$, where $T_{\text{bisect}}(n, m)$ denotes the cost

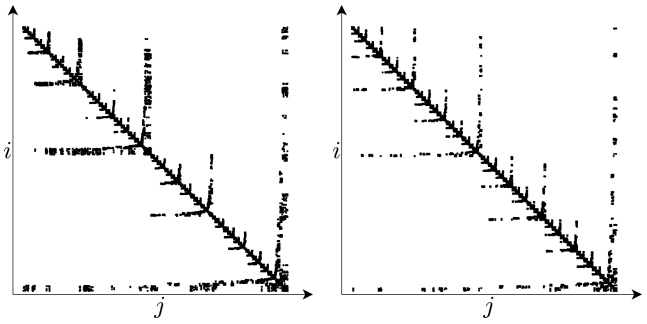
of a single graph bisection. The overall process is outlined in Algorithm 7.

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 [3]. Nested dissection can be implemented using the partitioning methods presented in Section 3.1 and Section 3.2 for computing the node separator. Figure 8 illustrates how different separator methods permute the adjacency matrix into a hierarchy of interior and separator blocks, yielding distinctive banded or clustered nonzero patterns. The differences between the separator methods are reflected in the number of nonzero entries in the Cholesky factor [14].



(a) Original adjacency, Cholesky nnz = 93892

(b) Coordinate bisection, Cholesky nnz = 89566



(c) Inertial bisection, Cholesky nnz = 89653

(d) Spectral bisection, Cholesky nnz = 89110

Fig. 8: Comparison of the (a) original adjacency matrix of the `3elt` [9] graph against three nested dissection reoderings obtained using (b) coordinate, (c) inertial, and (d) spectral-based separators. Number of nonzeros (nnz) in the Cholesky factor are reported in the subcaptions.

The nested dissection provided by `GraphLab.jl` is called using:

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

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

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

4. Framework and tools for graph bisection

`GraphLab.jl` provides a framework for graph partitioning that consists of the following core modules:

(1) **Graph creation**: Generating graphs with node positions defined by coordinate systems.
(2) **Graph partitioning**: Implementing the recursive bisection methods detailed in Section 3.
(3) **Benchmarking**: Measuring and comparing algorithm performance according to graph cut criteria.
(4) **Visualization**: Visualizing graphs and their partitions to facilitate analysis and comparison.

4.1 Graph creation

Graphs can either be synthetically 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 coordinates, or a `.csv` file with coordinates only, from which the adjacency matrix is constructed using k-nearest neighbors using `NearestNeighbors.jl`[1], with a default number of 5 neighbors. The file parsing and data loading are handled by the external packages `MAT.jl`[2] and `CSV.jl`[3]

4.2 Benchmarking

The framework includes two example scripts, provided in the `examples` directory of the `GraphLab.jl` package, that benchmark the implemented graph partitioning methods in terms of edge, ratio, and normalized cut, and in terms of node balance ratio. The edge cut measures the number of edges crossing between partitions. For a k-way partitiong $\mathcal{V}_1, \ldots, \mathcal{V}_k$, it is defined as $\text{cut}(\mathcal{V}_1, \ldots, \mathcal{V}_k) = \left|\{(u, v) \in E : u \in \mathcal{V}_i, v \in \mathcal{V}_j, i \neq j\}\right|$. The balance ratio quantifies the distribution of vertices among partitions. Let n be the total

[1] `https://github.com/KristofferC/NearestNeighbors.jl`
[2] `https://github.com/JuliaIO/MAT.jl`
[3] `https://github.com/JuliaData/CSV.jl`

number of vertices, k the number of partitions, and $|\mathcal{V}_i|$ the size of partition $\mathcal{V}_i$. The ideal partition size is n/k, and the node balance ratio is defined as $\text{bal} = \frac{\max_i |\mathcal{V}_i|}{n/k}$. A perfectly balanced partitioning results in $\text{bal} = 1$, while larger values indicate increasing imbalance. To evaluate both separation quality and partition balance, two common normalized metrics are used. The Normalized Cut is defined as $\text{NCut} = \sum_{i=1}^{k} \frac{\text{cut}(\mathcal{V}_i, \bar{\mathcal{V}}_i)}{\text{vol}(\mathcal{V}_i)}$, where $\text{vol}(\mathcal{V}_i) = \sum_{v \in \mathcal{V}_i} \deg(v)$ and $\bar{\mathcal{V}}_i$ denotes the complement of $\mathcal{V}_i$ in $\mathcal{V}$, i.e., $\bar{\mathcal{V}}_i = \mathcal{V} \setminus \mathcal{V}_i$. The Ratio Cut is defined as $\text{RCut} = \sum_{i=1}^{k} \frac{\text{cut}(\mathcal{V}_i, \bar{\mathcal{V}}_i)}{|\mathcal{V}_i|}$. Both metrics penalize unbalanced partitions and favor cuts that correspond to sparse inter-partition connections in the graph. The provided benchmarking example scripts are:

(1) `ex1.jl`: Benchmarks multiple bisection methods across a set of mesh inputs from the university of Florida sparse matrix collection [9]. For each method and mesh, it computes the edge cut, normalized cut, and ratio cut. As shown in Table 1, no single bisection strategy is consistently best across all considered meshes.

(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 node balance are recorded for each case.

4.3 Visualization

To visualize graph partitions, the adjacency matrix $\mathbf{A}$, the vertex coordinates, and the corresponding partitioning are required. `GraphLab.jl` integrates several Julia packages to support this process. Our visualization routine utilizes the edge-drawing logic from the visualization code available in the `SGtSNEpi.jl` repository [26]. `Graphs.jl` [11] supplies fundamental graph operations and data structures for computations on the partitioned graphs, and `Makie.jl` [8] enables the creation of customizable plots with visually distinct color palettes from `Colors.jl` [4].
Illustrations of selected bipartitioning and recursive bisection results are offered in Figure 9 and Figure 10, respectively.

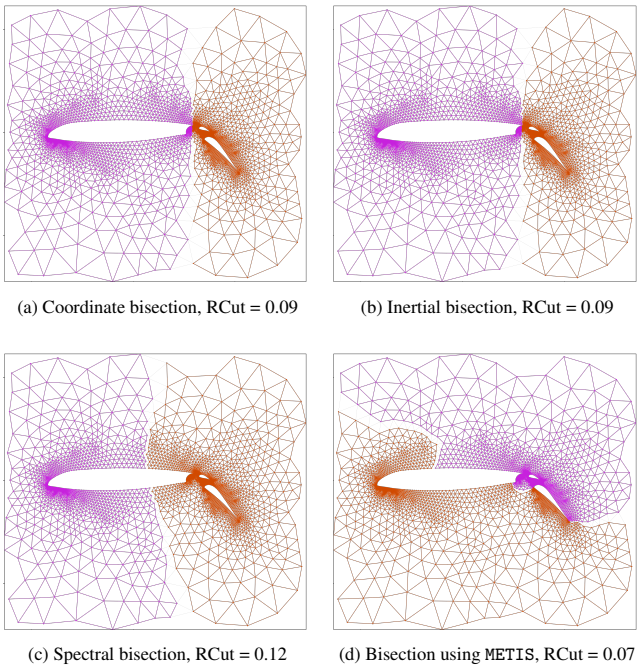
5. Installation and demonstration

To install the package from GitHub[6] 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")
```

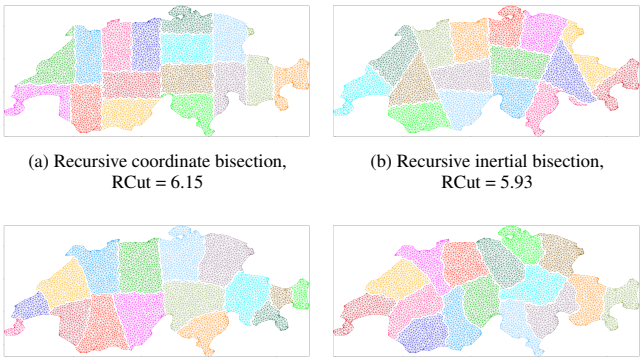
(2) As a basic example for graph partitioning, the adjacency matrix $\mathbf{A}$ and vertex coordinates are first constructed from the input data. Here, we generate a synthetic 10×50 rectangular grid graph rotated by an angle of $\pi/3$ radians. Spectral bisection is then applied to compute the partition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



(a) Coordinate bisection, RCut = 0.09 (b) Inertial bisection, RCut = 0.09

(c) Spectral bisection, RCut = 0.12 (d) Bisection using `METIS`, RCut = 0.07

Fig. 9: Visualization of four graph bisection methods applied to the `airfoil1` mesh (4253 nodes and 12289 edges) [9], illustrating the different partitioning results and the associated ratio cut.



(a) Recursive coordinate bisection, RCut = 6.15 (b) Recursive inertial bisection, RCut = 5.93

(c) Recursive spectral bisection, RCut = 5.39 (d) Recursive `METIS` bisection, RCut = 5.01

Fig. 10: Comparison of four graph recursive bisection methods applied to the `Swiss_graph`[5] (4468 nodes and 15230 edges), illustrating differences in the 16-way recursive partitioning and the associated ratio cut.

```
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")
```

Further details about the package and its functionalities can be found in the online documentation[7].

[4] `https://juliagraphics.github.io/Colors.jl`
[5] `https://github.com/lechekhabm/GraphLab.jl/tree/main/examples/meshes`
[6] `https://github.com/lechekhabm/GraphLab.jl`
[7] `https://lechekhabm.github.io/GraphLab.jl/dev/`

Table 1. : Edge cuts (EC), normalized cuts (NC), and ratio cuts (RC) for each method and mesh.

Mesh	Coordinate			Inertial			Random sphere			Adaptive SFC			Spectral			Geo+Spectral		
	EC	*NC*	*RC*	*EC*	*NC*	*RC*	*EC*	*NC*	*RC*	*EC*	*NC*	*RC*	*EC*	*NC*	*RC*	*EC*	*NC*	*RC*
`3elt`	172	0.03	0.15	209	0.03	0.18	101	0.02	0.09	224	0.03	0.19	117	0.02	0.10	117	0.02	0.10
`airfoil1`	94	0.02	0.09	94	0.02	0.09	93	0.02	0.09	98	0.02	0.09	132	0.02	0.12	132	0.02	0.12
`barth4`	206	0.02	0.14	194	0.02	0.13	130	0.01	0.09	208	0.02	0.14	127	0.01	0.08	127	0.01	0.08
`crack`	323	0.02	0.13	377	0.03	0.15	274	0.02	0.11	353	0.02	0.14	233	0.02	0.09	233	0.02	0.09
`mesh1e1`	18	0.24	1.50	19	0.25	1.58	17	0.24	1.50	18	0.24	1.50	18	0.24	1.50	18	0.24	1.50
`mesh2e1`	37	0.07	0.48	47	0.09	0.61	35	0.07	0.46	40	0.08	0.52	35	0.07	0.46	35	0.07	0.46
`mesh3e1`	17	0.05	0.24	32	0.09	0.44	18	0.06	0.25	21	0.06	0.29	30	0.09	0.42	20	0.05	0.36
`mesh3em5`	17	0.05	0.24	32	0.09	0.44	18	0.06	0.25	21	0.06	0.29	18	0.05	0.25	19	0.05	0.25
`netz4504_dual`	25	0.04	0.16	30	0.05	0.20	24	0.04	0.15	25	0.04	0.16	23	0.04	0.15	23	0.04	0.15
`stufe`	16	0.02	0.06	16	0.02	0.06	16	0.02	0.06	16	0.02	0.06	16	0.02	0.06	16	0.02	0.06
`ukerbe1`	27	0.01	0.02	28	0.01	0.02	37	0.01	0.02	34	0.01	0.02	29	0.01	0.02	28	0.01	0.02

6. Conclusion and future works

In this work, we have presented `GraphLab.jl`, a framework for graph partitioning designed to support research and education in graph theory and partitioning problems. It incorporates fundamental partitioning methods, such as coordinate, inertial, and spectral bisection, as well as random spheres, space-filling curves, and nested dissection. It additionally offers utilities for visualization, benchmarking, and partition quality assessment, in an effort to provide a unified environment for analyzing and comparing graph partitioning algorithms. Ongoing developments aim to broaden the framework's capabilities with implementations of additional partitioning techniques, of multilevel methods [21, 28], and by enabling parallel execution [23], particularly in the recursive implementation of geometric-based algorithms.

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] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. HipMCL: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks. *Nucleic Acids Research*, 46(6):e33–e33, 01 2018. doi:10.1093/nar/gkx1313.

[2] Charles-Edmond Bichot. *General Introduction to Graph Partitioning*, chapter 1, pages 1–25. John Wiley & Sons, Ltd, 2013. doi:10.1002/9781118601181.ch1.

[3] Matthias Bollhöfer, Olaf Schenk, Radim Janalik, Steve Hamm, and Kiran Gullapalli. *State-of-the-Art Sparse Direct Solvers*, pages 3–33. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-43736-7_1.

[4] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. *Graph theory with applications*, volume 290. Macmillan London, 1976.

[5] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent Advances in Graph Partitioning*, pages 117–158. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-49487-6_4.

[6] 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. doi:10.1145/3571808.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[8] Simon Danisch and Julius Krumbiegel. Makie.jl: Flexible high-performance data visualization for Julia. *Journal of Open Source Software*, 6(65):3349, 2021. doi:10.21105/joss.03349.

[9] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011. doi:10.1145/2049662.2049663.

[10] Ulrich Elsner. Graph partitioning - a survey. Preprintreihe des Chemnitzer SFB 393 97-27, Technische Universität Chemnitz, Chemnitz, 1997.

[11] James Fairbanks, Mathieu Besançon, Schölly Simon, Júlio Hoffiman, Nick Eubank, and Stefan Karpinski. JuliaGraphs/Graphs.jl: an optimized graphs package for the Julia programming language, 2021.

[12] 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. doi:10.1002/nme.1620360503.

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

[14] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973. doi:10.1137/0710032.

[15] John R. Gilbert, Yingzhou Li, and Shang-Hua Teng. Mesh partitioning toolbox – meshpart, April 2020. doi:10.5281/zenodo.3746723. `https://github.com/YingzhouLi/meshpart`.

[16] 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. doi:10.1137/S1064827594275339.

[17] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013.

[18] 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. doi:10.1137/S1064827595287997.

[19] Brian Wilson Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970. doi:10.1002/j.1538-7305.1970.tb01770.x.

[20] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45(4):255, October 1950. doi:10.6028/jres.045.026.

[21] Malik Lechekhab, Dimosthenis Pasadakis, and Olaf Schenk. Multilevel diffusion based spectral graph clustering. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2024. doi:10.1109/HPEC62836.2024.10938528.

[22] Richard B. Lehoucq, Danny C. Sorensen, and Chao Yang. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, volume 6 of *Software, Environments, and Tools*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998. doi:10.1137/1.9780898719628.

[23] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2625–2638, 2017. doi:10.1109/TPDS.2017.2671868.

[24] M. E. J. Newman. Community detection and graph partitioning. *EPL (Europhysics Letters)*, 103(2):28003, July 2013. doi:10.1209/0295-5075/103/28003.

[25] Dimosthenis Pasadakis, Matthias Bollhöfer, and Olaf Schenk. Sparse quadratic approximation for graph learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(9):11256–11269, 2023. doi:10.1109/TPAMI.2023.3263969.

[26] Nikos Pitsianis, Alexandros-Stavros Iliopoulos, Dimitris Floros, and Xiaobai Sun. Spaceland Embedding of Sparse Stochastic Graphs. In *IEEE High Performance Extreme Computing Conference*, 11 2019. doi:10.1109/HPEC.2019.8916505.

[27] Giulio Rossetti, Letizia Milli, and Rémy Cazabet. CDLIB: a python library to extract, compare and evaluate communities from complex networks. *Applied Network Science*, 4(1):52, 2019. doi:10.1007/s41109-019-0165-9.

[28] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933, pages 164–175. Springer, 2013. doi:10.48550/arXiv.1210.0477.

[29] 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. doi:10.1109/HPCC-CSS-ICESS.2015.192.

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

[31] Horst D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2):135–148, 1991. doi:10.1016/0956-0521(91)90014-V.

[32] Horst D. Simon and Shang-Hua Teng. How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, 1997. doi:10.1137/S1064827593255135.

[33] James A. Storer. *An Introduction to Data Structures and Algorithms*. Birkhäuser, Boston, MA, 2002. doi:10.1007/978-1-4612-0075-8.