Università della Svizzera italiana
Institute of Computing CI

Numerical Computing **2024**

Student: Matteo Ghilardini **Discussed with:** Sasha Toscano

Solution for Project 3 **Due date:** Wednesday, 13 November 2024, 11:59 PM

1. Implementing various graph partitioning algorithms [50 points]

- **Analysis of** `Bench_bisection.m`
  This script is used to analyze an compare the performaces of various bisection methods acros different graphs/meshes. It starts loadign all the meshes/graphs and plotting them. After that runs the bisection algorithms on each of them and plots the results showing also the number of *cut edges*.
  The bisection algorithms used are:
  - `Coordinate bisection`: it is a simple bisection algorithm that splits the graph in two parts based on the spatial layout of the nodes.
  - `Metis bisection`: it uses the Metis library to partition the graph in two parts focusing on *edge-cut minimization*.
  - `Spectral bisection`: it uses the Fiedler vector to partition the graph in two balanced parts.
  - `Inertial bisection`: it uses the Fiedler vector to partition the graph in two parts relying on the geometric layout of the graph.

- **Implementation of spectral graph bisection in script** `bisection_spectral.m`[1]
  Following the steps reported in the script, we do the following:
  1. **Construct the Laplacian matrix:** L of the graph by subtracting the adjacency matrix A from the degree matrix D. The degree matrix D is computed as the diagonal matrix of the sum of the adjacency matrix rows (`diag(sum(A, 2))`).
  2. **Calculate its eigensdecomposition:** extract the eigenvector with the second smallest real eigenvalue in L (i.e. Fiedler eigenvector[2]) using the formula `[W, ~] = eigs(L, 2, 'smallestreal');`.
  3. **Label the vertices with the components of the Fiedler vector:** takes out the second smallest eigenvector of W corresponding to the second column of the W matrix and assign it to $w2$.

[1] This script `bisection_spectral.m` don't run by itself since requires some parameters. To test/run it, run `Bench_bisection.m`

[2] The **Fiedler vector** is the eigenvector associated with the second smallest eigenvalue of the Laplacian matrix of a graph. It is important in graph partitioning because it provides a way to partition the graph into balanced parts

4. **Partition them around their median value, or 0:** we set by default the threshold to the median of `vertexLabels` (if want to use 0 as threshold, simply comment or uncomment the lines) and create 2 partitions based on the threshold.
   By thresholding the Fiedler vector around 0, we partition the graph based on the sign of the components of the Fiedler vector. The vertices with positive components are assigned to one partition, while the vertices with negative components are assigned to the other partition.
   Otherwise thresholding the Fiedler vector around the median value, we partition the graph based on the magnitude of the components of the Fiedler vector. The vertices with components greater than the median are assigned to one partition, while the vertices with components less than the median are assigned to the other partition.
   In general, thresholding at 0 results in two roughly balanced (equal sized) partitions with minimum edgecut, while thresholding produces two strictly balanced partitions.

   The difference in terms of edgecuts for our meshes using the different thresholds is following:

Mesh	Spectral (t=median)	Spectral (t=0)
grid5rec(12,100)	12	12
grid5rec(100,12)	12	12
grid5recRotate(100,12,-45)	12	12
gridt(50)	76	68
grid9(40)	140	140
Smallmesh	14	12
Tapir	58	18
Eppstein	47	42

Table 1: Edgecuts for spectral bisection with different thresholds

From the table 1 we can see that the thresholding at 0 actually result in a better edgecut for some of the meshes. In any case, for general implementation, I will use the thresholding at the median value of the Fiedler vector to ensure a strictly balanced partitioning.

- **Implementation of inertial graph bisection in script `bisection_inertial.m`[3]**
  Following the steps reported in the script, we do the following:
  1. **Calculate the center of mass:** compute the number of vertices in xy using `n = size(xy, 1)` and then compute the center of mass as the median of each column in xy using `center_of_mass = mean(xy, 1)`
  2. **Construct the matrix M:** compute all elements of M following the formula given in the pdf of the assignment and compose $M = \begin{bmatrix} Sxx & Sxy \\ Sxy & Syy \end{bmatrix}$
  3. **Calculate the smallest eigenvector of M:** extract the eigenvector with the smallest real eigenvalue in M using theformula `[eigv, ~] = eigs(M, 1, "smallestabs")`.
  4. **Find the line L on which the center of mass lies:** create the line L by swapping and flipping the components of $eigv$ to ensure it is perpendicular to the center of mass and make it as a unit-vector by deviding it by its norm.
  5. **Partition the points around the line L:** partition the points in xy using the given function `partition(...)` using the line L as the partitioning line.

[3]This script `bisection_inertial.m` don't run by itself since requires some parameters. To test/run it, run `Bench_bisection.m`

- **Bisection results[4]:** in the following table we report the number of edgecut for all the provided meshes using the different bisection algorithms:

Mesh	Coordinate	Metis 5.0.2	Spectral	Inertial
grid5rec(12,100)	12	14	12	12
grid5rec(100,12)	12	12	12	12
grid5recRotate(100,12,-45)	22	12	12	12
gridt(50)	72	76	76	72
grid9(40)	118	129	140	118
Smallmesh	25	13	14	30
Tapir	55	34	58	49
Eppstein	42	48	47	45

Table 2: Bisection results

From the table 2 we can see that there is not an absolute "best" bisection algorithm for all the meshes. The Metis bisection algorithm is the best in terms of edgecuts for most of the meshes, but not for all of them (for example for `gridt(50)`, `grid9(40)`, or `Eppstein`).
This shows the important of chosing the right bisection algorithm based on the characteristics of the mesh/graph.

2. Recursively bisecting meshes [20 points]

The implementation of the recursive bisection is done in the script `Bench_rec_bisection.m` and the fill-out portions of code are pretty straightforward.

After loading each mesh, the script call the function `rec_bisection(...)` for all the bisection algorithms passing as parameters the function itself, the levels of recursion, and then the adjacency matrix and the coordinates of the mesh. The function `rec_bisection(...)` is a recursive function that at each level of recursion calls the bisection algorithm on the mesh and then splits the mesh in two parts. The recursion stops when the number of partitions is equal to the number of levels of recursion.
After it is counted the number of edgecuts and printed the results in the tables below.
To make the code more usable, i added also the possibility to choose if want to see the graphs of the partitions or not for each mesh.

Case	Size [nodes, edges]	Spectral	Metis	Coordinate	Intertial
mesh1e1	[48, 129]	58	55	63	59
bodyy4	[17546, 52002]	4913	965	1065	1363
de2010	[24115, 58028]	4227	525	929	1080
biplane-9	[21701, 42038]	511	467	548	647
L-9	[17983, 35596]	704	648	631	828

Table 3: Recursive bisection results for $2^3 = 8$ partitions

From the table 3, can be seen that the Metis bisection algorithm is the best in terms of edgecuts for all the meshes.Otherwise, the spectral bisection algorithm is the worst in terms of edgecuts for more or less all the meshes.
Can also be noticed that the edgecuts in the smallest mesh (`mesh1e1`) are all pretty similar, among all the bisection algorithms.

The first thing that can be noticed is that also in table 4 the Metis bisection algorithm is the best in terms of edgecuts for all the meshes.
Can also be noticed that also in this case, the edgecuts in the smallest mesh (`mesh1e1`) are all

[4]The script to run to get the results is `Bench_bisection.m`

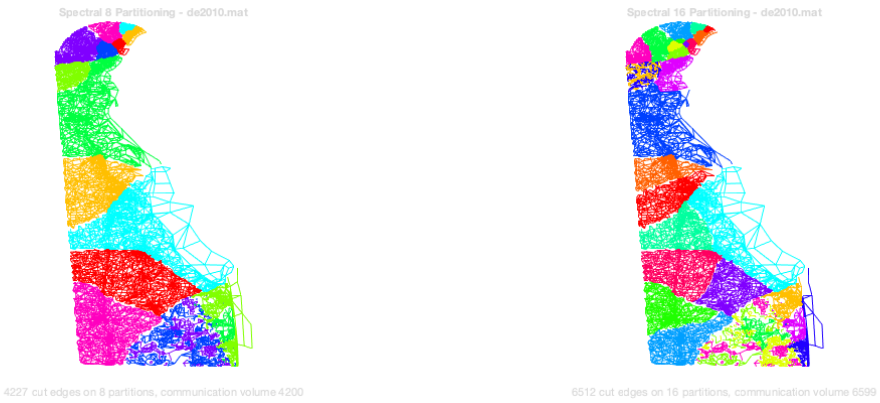
Case	Size [nodes, edges]	Spectral	Metis	Coordinate	Intertial
mesh1e1	[48, 129]	58	55	63	59
bodyy4	[17546, 52002]	7311	1620	1951	2212
de2010	[24115, 58028]	6512	952	1796	1996
biplane-9	[21701, 42038]	897	831	974	1092
L-9	[17983, 35596]	1122	1016	1028	1378

Table 4: Recursive bisection results for $2^4 = 16$ partitions

pretty similar, among all the bisection algorithms.
An interesting thing to notice is that the edgecuts for the `mesh1e1` mesh are identical between the table 3 and the table 4. This can be explained by the fact that the mesh is very small and so there are fewer opportunities to create additional edge cuts as the number of partitions increases.

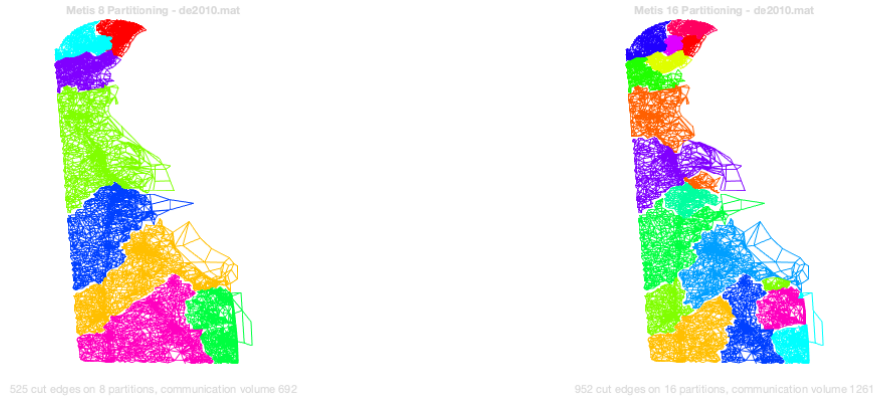
In general terms we see in both tables that the Metis bisection algorithm is the best in terms of edgecuts for all the meshes since it is the one that minimizes the number of edgecuts.

An example of the resulting partitioning, it follows the figures for the `de-2010` mesh with 8 and 16 partitions:



(a) 8 partitions (b) 16 partitions

Figure 1: Spectral partition of the `de-2010` mesh



(a) 8 partitions (b) 16 partitions

Figure 2: Metis partition of the `de-2010` mesh

(a) 8 partitions (b) 16 partitions

Figure 3: Coordinate partition of the `de-2010` mesh

(a) 8 partitions (b) 16 partitions

Figure 4: Inertial partition of the `de-2010` mesh

Comparing the resulting figures of the "best" (*Metis*) and the "worst" (*Spectral*) algorithms, we can see that the *Metis* bisection algorithm 2 creates more "shaped" partitions compared to the *Spectral* bisection algorithm. This can be easily seen in the lower part of the figures 1a and 1b where the *Spectral* bisection algorithm creates a lot "thin" partitions respect to the figures 2a and 2b where the *Metis* bisection algorithm creates more "compact" partitions. This is probably due to the fact that the *Metis* bisection algorithm tries to minimize the number of edgecuts.

We also note that the other bisection algorithms, *Coordinate* (figure 3) and *Inertial* (Figure 4), create partitions with edge cuts generally similar to those produced by Metis, but with different shapes.

In particular, the *Coordinate* algorithm (figure 3) tends to produce more "square" partitions, as it is based on the spatial positions of the nodes.

The *Inertial* algorithm (figure 4), on the other hand, tends to create more "elongated" and sometimes "triangular" partitions, due to its reliance on the geometric and directional distribution of the nodes.

The main difference between the *Coordinate* and *Inertial* approaches is that *Coordinate* relies on the absolute spatial positions of nodes, while *Inertial* is based on how nodes are connected to each other.

3. Comparing recursive bisection to direct k-way partitioning [15 points]

Considering the key difference between recursive bisection and k-way partitioning, we expect for the *Helicopter*, since is quite "small" as a mesh, the recursive bisection can performe better in terms of edgecuts. Differently for the *Skirt* mesh, that is "bigger" we expect that the k-way partitioning should be better.
Considering that the recursive bisection algorithm is better in terms of edgecuts when the mesh is more "regular" and the *Helicopter* mesh is quite "complex" in terms of shape, the results can differe from the expectations.

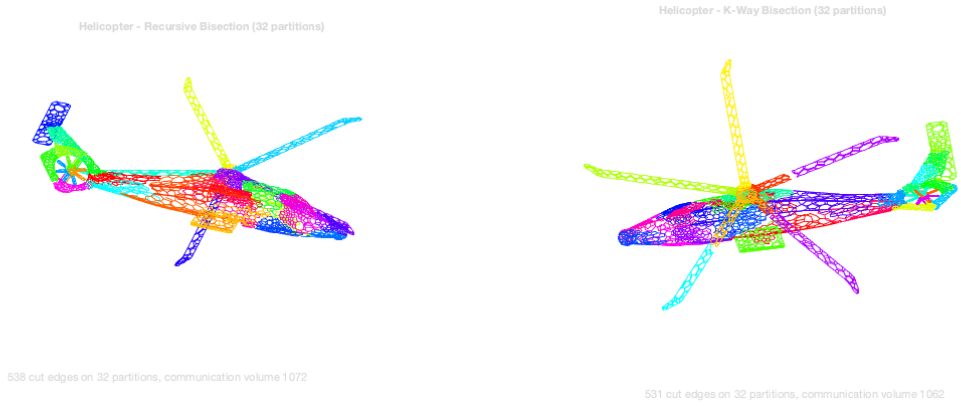
The implementation for retrieve the number of edgecut is done in the script `Bench_metis.m` and the fill-out portions of code are pretty straightforward.
The script loads both the meshes for *Helicopter* and *Skirt* and then runs the Metis partition using the function `metismex(...)` passing the partition type (respectively *PartGraphRecursive* or *PartGraphKway*), the initial adjacency matrix (for both *Helicopter* and *Skirt*) and the number of partitions.
After each run of `metismex(...)` the resulting map and edgecuts are stored in variables and then printed giving out the following results:

Partitions	**helicopter (7920 nodes)**	**skirt (12598)**
16-recursive bisection	328	3112
16-way direct partition	341	3350
32-recursive bisection	538	6345
32-way direct partition	531	6284

Table 5: Comparing the number of cut edges for recursive bisection and direct multiway partitioning in Metis

From the table 5 we can see that the recursive bisection algorithm is better in terms of edgecuts for the *Helicopter* mesh, while the direct k-way partitioning is better for the *Skirt* mesh.
This can be explained by the fact that the recursive bisection algorithm is better in terms of edgecuts when the mesh is more "regular" and "compact" like the *Helicopter* mesh, while the direct k-way partitioning is better when the mesh is more "irregular" and "sparse" like the *Skirt* mesh.

The resulting figures shows the partitioning of the *Helicopter* and *Skirt* meshes with 32 partitions:



(a) Recursive bisection (b) K-way partitioning

Figure 5: Partitioning of the *Helicopter* mesh with 32 partitions

(a) Recursive bisection (b) K-way partitioning

Figure 6: Partitioning of the *Skirt* mesh with 32 partitions

As seen also in table 5, the recursive bisection algorithm (compare figures 5a and 5b) is better in terms of edgecuts for the *Helicopter* mesh, while the direct k-way partitioning is better for the *Skirt* mesh (compare figures 6a and 6b).
In both cases (*Helicopter* 5, and *Skirt* 6), the *communication volume* (so the number of distinct vertices are connected to vertices in another partition) is more or less the same for both algorithms.

The fact that the *Skirt* 6 mesh has arount "10 times" more edgecuts, and the double of communication volume than the *Helicopter* 5 mesh also if it's size is only around "1.5 times" bigger, can be explained by the fact that the *Skirt* mesh is more "irregular" and "sparse" than the *Helicopter* mesh. This means that the *Skirt* mesh has more "long" and "thin" partitions, which require more edgecuts to connect them.