NRGG – Neighbour Restricting Graph Grow Partitioning Library 1.0
User Guide

Yadong Xu
March, 2017
TUMCREATE Ltd, Singapore
Nanyang Technological University, Singapore
xuya0006@e.ntu.edu.sg

Contents

1. Overview of NRGG

NRGG, namely, neighbour-restricting graph grow partitioning algorithm is particularly designed for partitioning ***2D spatial graphs***, where the vertices have two-dimensional Cartesian coordinates. It considers three optimization objectives: minimizing neighbouring partitions, minimizing edge cut between partitions, minimizing imbalance of vertex distribution. There are trade-offs between these objectives. This NRGG algorithm prioritizes on **minimizing neighbouring partitions** compared to minimizing edge cut. This is the major difference between NRGG and the well-known METIS partitioning program.

The algorithm consists of two phases: graph-grow phase, and refinement phase. In the graph-grow phase, starting from an initial vertex, subgraphs are grown one by one along the edges of the graph. It can be easily proven that for a 2D space, cutting the space into stripes generates the smallest number of neighbouring partitions. In order to limit the number of neighbouring partitions, we apply this idea for partitioning a graph, i.e., generate stripe-like partitions.

After the initial partitioning, in the refinement phase, partitions are refined in order to reduce edge cut, as well as to alleviate workload imbalance in case there is severe imbalance. local search refinement heuristic similar to Karypis and Kumar's boundary refinement heuristic [1], which is a variation of the original heuristic of Kernighan and Lin [2]. A distinct feature of our local search refinement algorithm is to restrict neighbouring partitions.

For more detailed explanation, user can refer to our published work [3]. In that work, NRGG is used in partitioning road networks for parallel road traffic simulation.

The current version of this library provides mainly has two different options: i) just use NRGG; and ii) a multi-level NRGG. In the multi-level approach, the input graph is first coarsened recursively to a small graph, whose size of specified by the user. Then partitioning is applied to the coarsest graph. Then the partitioned graph is uncoarsened back to the original graph.

2. NRGG API

Any program that uses NRGG's API needs to include the *nrgg.h* header file. The API designed here is also similar to METIS' for easy use. This section only describes the API interface. The input data structure also closely follows METIS' format, which will be described in the next section.

The basic data types we used here are *int* and *double*. Data structures, such as *unordered_map*, *unordered_set*, *map*, *priority_queue*, *variable array* (*vector*), are all from the standard C++ library.

Currently, it only supports C++. Of course, users can use *JNI* to use it for Java programs, however, that won't be provided by the current version.

int NRGG_PartGraph(int* *nvtxs*, int* *vwgt*, double* *vcrdnt*, int* *xadj*, int* *adjncy*, int* *adjwgt*, int* *nparts*, int* *part_out*, int* *edge_out*, bool *wcout*, double* *imb_thres=NULL*, int* *rf_itr=NULL*, bool *multilvl=false*, int* *cstvtx=NULL*);

Parameters

nvtxs	the number of vertices in the graph.
vwgt	the weights of vertices ordered by vertex *id* It must have size *nvtxs*. If *vwgt=NULL*, then all vertices will be assigned with the same weight.
vcrdnt	the coordinates of vertices ordered by vertex *id* It must have a size of *2nvtxs*.
xadj	the pointer to the adjacency lists of the vertices. The array should have a size of *nvtxs + 1*.
adjncy	the pointer to the adjacency array which holds the adjacency lists of the vertices. The array should have a size of *2m*.
adjwgt	The weights of edges
nparts	the number of blocks you want the graph to be partitioned in.
part_out	the output of the partitioning algorithm on partition ids of vertices It has to be a pre-allocated *int* array of size *nvtxs*. After the function call this array contains the resultant partitioning information of the vertices. The *i*th vertex is assigned to *part*[*i*].
edge_out	the output on the total weight of edges cut between partitions
wcout (optional)	whether detailed partitioning traces are printed with std::cout Default=*true*.
imb_thres (optional)	thresholds of allowing imbalance of weights of partitions during refinement *imb_thres*[*0*] is the low threshold, and *imb_thres*[*1*] is the hight threshold. If *imb_thres = NULL*, then default low threshold=0.9*average weight, high threshold=1.02* average weight.
rf_itr (optional)	the maximum number of iterations allowed in refinement phase If *rf_itr=NULL*, then use the default value *8*.
multilvl (optional)	Whether to use multi-level partitioning, *heavy edge matching* will be used. The NRGG algorithm will be applied on the coarsest graph.
cstvtx (optional)	The maximum number of vertices in the coarsest graph. E.g., if *cstvtx=100, each partition in the coarsest graph will have less than 100 vertices. Only used when *multilvl* is set to *true*. Default value is 100.

The *int return value* is an indicator whether the algorithm has executed successfully. If yes, return 1; otherwise, return 0.

If you are not sure whether to use the last five parameters, you can simply ignore them.

3. Graph Data Structure

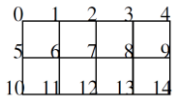
This sections lists the format of input data required in the *NRGG_PartGraph()* function. The formats for vertex lists and adjacency lists are exactly the same as the one used by METIS, except the extra spatial information of vertices. This design is for easy application of this library for the programs that are already using METIS.

Following texts are snipped from METIS version 5.1.0 user manual.

(snip start)

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

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



(a) A sample graph

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

(b) CSR format

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

The weights of the vertices (if any) are stored in an additional array called *vwgt*. If *ncon* is the number of weights associated with each vertex, the array *vwgt* contains *n* * *ncon* elements (recall that n is the number of vertices). The weights of the *i*th vertex are stored in *ncon* consecutive entries starting at location *vwgt*[*i* * ncon]. Note that if each vertex has only a single weight, then *vwgt* will contain *n* elements, and *vwgt*[*i*] will store the weight of the *i*th vertex. The vertex-weights must be integers greater or equal to zero. If all the vertices of the graph have the same weight (i.e., the graph is unweighted), then the *vwgt* can be set to NULL. The weights of the edges (if any) are stored in an additional array called *adjwgt*. This array contains *2m* elements, and the weight of edge *adjncy*[*j*] is stored at location *adjwgt*[*j*]. The

edge-weights must be integers greater than zero. If all the edges of the graph have the same weight (i.e., the graph is unweighted), then the *adjwgt* can be set to NULL.

(snip end)

The number of vertices *n* is the *nvtxs* parameter in the NRGG API. Compared to the input required by METIS, NRGG need extra information on the coordinates of the vertices. It should be provided in the array *vcrdnt*. Vcrdnt must have a size of *2*n*. The (*2*i*)th element is the *x* coordinate of the *i*th vertex, and the (*2*i+1*)th element is the *y* coordinate of the *i*th vertex. In this version, the graph will be cut along the *x* coordinate.

References

[1] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Jan. 1998.

[2] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Syst. Tech. J.*, vol. 49, no. 1, pp. 291–307, 1970.

[3] Y. Xu, W. Cai, D. Eckhoff, S. Nair, and A. Knoll, "A Graph Partitioning Algorithm for Parallel Agent-Based Road Traffic Simulation," in *2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, 2017.