LAGraph

Tim Davis, Tim Mattson, Scott McMillan, Jim Kitchen, Erik Welch

Jun 03, 2025

CONTENTS:

The LAGraph library is a collection of high level graph algorithms based on the GraphBLAS C API. These algorithms construct graph algorithms expressed *in the language of linear algebra*. Graphs are expressed as matrices, and the operations over these matrices are generalized through the use of a semiring algebraic structure.

LAGraph is available at https://github.com/GraphBLAS/LAGraph. LAGraph requires SuiteSparse:GraphBLAS, available at https://github.com/DrTimothyAldenDavis/GraphBLAS.

CHAPTER
ONE

INTRODUCTION

A graph is a set of vertices and a set of edges between them. This pair of sets leads directly to the familiar picture of a graph as a set of dots connected by arcs (an undirected graphs) or arrows (a directed graph). You can also represent a graph in terms of matrices. Usually, this is done with an adjacency matrix where the rows and columns correspond to the vertices and the non-empty elements represent the edges between vertices. Since fully connected graphs (i.e., every vertex is connected to every other vertex) are rare, matrices used for Graphs are typically sparse (most elements are empty so it makes sense to only store the non-empty elements).

Representing a graph as a sparse matrix results in graph algorithms expressed in terms of linear algebra. For example, if a vector represents a set of vertices, multiplication of that vector by an adjacency matrix returns a vector of the neighbors of those vertices. A sequence of multiplications traverses the graph in a breadth first manner.

To cover the full range of graph algorithms, one additional ingredient is needed. We are used to thinking of matrix operations over real numbers: multiply pairs of matrix elements and then combine the resulting products through addition. There are times, however, when those operations do not provide the functionality needed by an algorithm. For example, it may be better to combine elements by only keeping the minimum value. The elements of the matrices may be Boolean values or integers or even a user-defined type. If the goal is to cover the full range of graph algorithms, therefore, we need a way to generalize the type and the operators to use instead of the usual addition and multiplication.

We do this through an algebraic semiring. This algebraic structure consists of (1) an operator corresponding to addition, (2) the identity of that operator, (3) an operator corresponding to multiplication, and (4) the identity of that operator. We are all familiar with the semiring used with real numbers consisting of (+,0,*,1). A common semiring in graph algorithms is the so-called tropical semiring consisting of (min,infinity,+,0). This is used in shortest path algorithms. These semirings give us a mathematically rigorous way to modify the operators used in our graph algorithms.

If you work with linear algebra, you most likely know about the Basic Linear Algebra subprograms or BLAS. Introduced in the 70's and 80's, the BLAS had a huge impact on the practice of linear algebra. By designing linear algebra in terms of the BLAS, an algorithm can be expressed at a high level leaving specialization to the low level details of a particular hardware platform to the BLAS. So if you want to use Linear Algebra for Graph Algorithms, it stands to reason that you need the Basic Linear Algebra Subprograms for Graph Algorithms. We call these the GraphBLAS (www.graphblas.org).

The GraphBLAS define opaque types for a matrix and a vector objects. Since these objects are opaque, an implementation has the freedom to specialize the data structures as needed to optimize the software for a particular platform. The GraphBLAS are great for people interested in sparse linear algebra and designing their own graph algorithms. The GraphBLAS library, however, does not include any graph algorithms. The GraphBLAS provide a software framework for constructing graph algorithms, but it doesn't provide any actual Graph Algorithms. Since most people working with graphs use algorithms but don't develop them "from scratch", the GraphBLAS are not really useful to most people.

Hence, there is a need for a library of Graph Algorithms implemented on top of the GraphBLAS. We have created this library. It is called LAGraph. The LAGraph library is a library of functions that implement the most common high level graph algorithms used in graph analytics. It includes types, utility functions and everything needed to incorporate graph algorithms into your analytics work flows. The library uses the GraphBLAS objects (e.g., GrB_matrix and GrB_vector) inside the objects defined by LAGraph. Consequently, GraphBLAS and LAGraph functions can be freely mixed inside a single program.

A graph in LAGraph uses the LAGraph_Graph data type. Unlike the GrB_matrix object, an LAGraph_Graph object is not opaque. The elements of the data structure are available to the user of the LAGraph library. The data associated with and LAGraph_Graph is represented by an GrB_matrix. The data structure includes information about the graph and key properties of the graph. For example, many algorithms require not only the matrix representing a graph, but also its transpose. These (and other) properties can be stored within the LAGraph_Graph. Storage of properties such as the transpose of a matrix requires additional storage, but the performance impact can more than compensate for the cost associated with that extra memory.

The algorithms within LAGraph roughly break down into two categories: Basic (`LAGraph_*`) and advanced (`LAGr_*`). The idea is that users who are not familiar with the ways graph algorithms are implemented and just want to apply an algorithm to their graphs, would use the Basic interface. For advanced users who are comfortable working with key aspects of the algorithms they are working with might see a significant performance benefit from working with the advanced algorithm.

For example, the basic and advanced algorithms deal with the properties of an LAGraph graph differently. The basic algorithm assumes the user will not set-up the LAGraph_Graph with the properties needed by an algorithm. Such properties will be computed as needed. An advanced user, however, may know that the string of operations in a workflow all requires a subset of key properties. By computing them in advanced and storing them with the LAGraph graph, the workflow can run much faster since it won't need to, for example, rearrange a matrix into its transpose for each algorithm in a workflow.

CHAPTER

TWO

LAGRAPH CONTEXT AND ERROR HANDLING

The sections below describe a set of functions that manage the LAGraph context within a user application, and discuss how errors are handled.

2.1 LAGraph Context Functions

int **`LAGraph_Init`**(char *msg)

LAGraph_Init: initializes GraphBLAS and LAGraph. This method must be called before calling any other GrB* or LAGraph* method. It initializes GraphBLAS with GrB_init and then performs LAGraph-specific initializations. In particular, the LAGraph semirings listed below are created. GrB_init can also safely be called before calling *LAGr_Init* or LAGraph_Init.

Parameters

`msg` – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_INVALID_VALUE`** – if LAGraph_Init or LAGr_Init has already been called by the user application.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGr_Init`**(GrB_Mode mode, void *(*user_malloc_function)(size_t), void *(*user_calloc_function)(size_t, size_t), void *(*user_realloc_function)(void*, size_t), void (*user_free_function)(void*), char *msg)

LAGr_Init: initializes GraphBLAS and LAGraph. LAGr_Init is identical to *LAGraph_Init* , except that it allows the user application to specify the GraphBLAS mode. It also provides four memory management functions, replacing the standard `malloc`, `calloc`, `realloc`, and `free`. The functions `user_malloc_function`, `user_calloc_function`, `user_realloc_function`, and `user_free_function` have the same signature as the ANSI C malloc, calloc, realloc, and free functions, respectively. Only user_malloc_function and user_free_function are required. user_calloc_function may be NULL, in which case `LAGraph_Calloc` uses `LAGraph_Malloc` and `memset`. Likewise, user_realloc_function may be NULL, in which case `LAGraph_Realloc` uses `LAGraph_Malloc`, `memcpy`, and `LAGraph_Free`.

Parameters

- **`mode`** – **[in]** the mode for GrB_Init
- **`user_malloc_function`** – **[in]** pointer to a malloc function
- **`user_calloc_function`** – **[in]** pointer to a calloc function, or NULL
- **`user_realloc_function`** – **[in]** pointer to a realalloc function, or NULL
- **`user_free_function`** – **[in]** pointer to a free function

- `msg` – **[inout]** any error messages.

Return values

- `GrB_SUCCESS` – if successful.
- `GrB_INVALID_VALUE` – if LAGraph_Init or LAGr_Init has already been called by the user application.
- `GrB_NULL_POINTER` – if user_malloc_function or user_free_function are NULL.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Finalize`**(char *msg)

LAGraph_Finalize: finish LAGraph and GraphBLAS. Must be called as the last LAGraph method. It calls GrB_finalize and frees any LAGraph objects created by *LAGraph_Init* or *LAGr_Init* . After calling this method, no LAGraph or GraphBLAS methods may be used.

Parameters

`msg` – **[inout]** any error messages.

Return values

`GrB_SUCCESS` – if successful.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Version`**(int version_number[3], char *version_date, char *msg)

LAGraph_Version: determines the version of LAGraph. The version number and date can also be obtained via compile-time constants from LAGraph.h. However, it is possible to compile a user application that includes one version of LAGraph.h and then links with another version of the LAGraph library later on, so the version number and date may differ from the compile-time constants.

The LAGraph_Version method allows the library itself to be queried, after it is linked in with the user application.

The version_number array is set to LAGRAPH_VERSION_MAJOR, LAGRAPH_VERSION_MINOR, and LAGRAPH_VERSION_UPDATE, in that order. The LAGRAPH_DATE string is copied into the user-provided version_date string, and is null-terminated.

Parameters

- `version_number` – **[out]** an array of size 3; with the major, minor, and update versions of LAGraph, in that order.
- `version_date` – **[out]** an array of size >= LAGraph_MSG_LEN, returned with the date of this version of LAGraph.
- `msg` – **[inout]** any error messages.

Return values

- `GrB_SUCCESS` – if successful.
- `GrB_NULL_POINTER` – if version_number or version_date are NULL.

int **`LAGraph_GetNumThreads`**(int *nthreads_outer, int *nthreads_inner, char *msg)

LAGraph_GetNumThreads determines the current number of OpenMP threads that can be used. See LAGraph_SetNumThreads for a description of nthreads_outer and nthreads_inner.

Parameters

- `nthreads_outer` – **[out]** number of threads for outer region.

- **`nthreads_inner`** – **[out]** number of threads for inner region, or for the underlying GraphBLAS library.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if nthreads_outer or nthreads_inner are NULL.

int **`LAGraph_SetNumThreads`**(int nthreads_outer, int nthreads_inner, char *msg)

LAGraph_SetNumThreads sets the current number of OpenMP threads that can be used by LAGraph and GraphBLAS. Two thread counts can be controlled:

Parameters

- **`nthreads_outer`** – **[in]** number of threads to be used in outer regions of a nested parallel construct assuming that nthreads_inner is used in the inner region. The total number of threads used for an entire nested region in LAGraph is given by nthreads_outer*nthreads_inner. This product is also the # of threads that a flat parallel region in LAGraph may use.
- **`nthreads_inner`** – **[in]** number of threads to be used in an inner region of a nested parallel construct, or for the # of threads to be used in each call to the underlying GraphBLAS library.
- **`msg`** – **[inout]** any error messages.

Return values

`GrB_SUCCESS` – if successful.

Returns

any GraphBLAS errors that may have been encountered.

2.2 Error handling

`LAGRAPH_RETURN_VALUES`

Nearly all LAGraph methods return an int to denote their status, and have a final string (msg) that captures any error messages.

LAGraph has a single function that does not follow this rule. *LAGraph_WallClockTime* has no error handling mechanism (it returns a value of type double, and does not have an final msg string parameter.

All other methods return an int to denote their status: zero if they are successful (which is the value of GrB_SUCCESS), negative on error, or positive for an informational value (such as GrB_NO_VALUE). Integers in the range -999 to 999 are reserved for GraphBLAS GrB_Info return values:

successful results:

- GrB_SUCCESS = 0 // all is well
- GrB_NO_VALUE = 1 // A(i,j) requested but not there

errors:

- GrB_UNINITIALIZED_OBJECT = -1 // object has not been initialized
- GrB_NULL_POINTER = -2 // input pointer is NULL

  - GrB_INVALID_VALUE = -3 // generic error; some value is bad
  - GrB_INVALID_INDEX = -4 // row or column index is out of bounds
  - GrB_DOMAIN_MISMATCH = -5 // object domains are not compatible
  - GrB_DIMENSION_MISMATCH = -6 // matrix dimensions do not match
  - GrB_OUTPUT_NOT_EMPTY = -7 // output matrix already has values
  - GrB_NOT_IMPLEMENTED = -8 // method not implemented
  - GrB_PANIC = -101 // unknown error
  - GrB_OUT_OF_MEMORY = -102 // out of memory
  - GrB_INSUFFICIENT_SPACE = -103, // output array not large enough
  - GrB_INVALID_OBJECT = -104 // object is corrupted
  - GrB_INDEX_OUT_OF_BOUNDS = -105 // row or col index out of bounds
  - GrB_EMPTY_OBJECT = -106 // an object does not contain a value

LAGraph returns any errors it receives from GraphBLAS, and also uses the GrB_* error codes in these cases:

- GrB_INVALID_INDEX: if a source node id is out of range
- GrB_INVALID_VALUE: if an enum to select an option is out of range
- GrB_NOT_IMPLEMENTED: if a type is not supported, or when SuiteSparse GraphBLAS is required.

Summary of return values for all LAGraph functions that return int:

- GrB_SUCCESS if successful
- a negative GrB_Info value on error (in range -999 to -1)
- a positive GrB_Info value if successful but with extra information (in range 1 to 999)
- -1999 to -1000: a common LAGraph-specific error, see list above
- 1000 to 1999: if successful, with extra LAGraph-specific information
- <= -2000: an LAGraph error specific to a particular LAGraph method
- >= 2000: an LAGraph warning specific to a particular LAGraph method

Many LAGraph methods share common error cases, described below. These return values are in the range -1000 to -1999. Return values of -2000 or greater may be used by specific LAGraph methods, which denote errors not in the following list:

- **LAGRAPH_INVALID_GRAPH (-1000):**
  The input graph is invalid; the details are given in the error msg string returned by the method.
- **LAGRAPH_SYMMETRIC_STRUCTURE_REQUIRED (-1001):**
  The method requires an undirected graph, or a directed graph with an adjacency matrix that is known to have a symmetric structure. LAGraph_Cached_IsSymmetricStructure can be used to determine this cached property.
- **LAGRAPH_IO_ERROR (-1002):**
  A file input or output method failed, or an input file has an incorrect format that cannot be parsed.
- **LAGRAPH_NOT_CACHED (-1003):**
  Some methods require one or more cached properties to be computed before calling them (AT, out_degree, or in_degree. Use LAGraph_Cached_AT, LAGraph_Cached_OutDegree, and/or LAGraph_Cached_InDegree to compute them.

- **LAGRAPH_NO_SELF_EDGES_ALLOWED (-1004):**
  Some methods requires that the graph have no self edges, which correspond to the entries on the diagonal of the adjacency matrix. If the G->nself_edges cached property is nonzero or unknown, this error condition is returned. Use LAGraph_Cached_NSelfEdges to compute G->nself_edges, or LAGraph_DeleteSelfEdges to remove all diagonal entries from G->A.
- **LAGRAPH_CONVERGENCE_FAILURE (-1005):**
  An iterative process failed to converge to a good solution.
- **LAGRAPH_CACHE_NOT_NEEDED (1000):**
  This is a warning, not an error. It is returned by LAGraph_Cached_* methods when asked to compute cached properties that are not needed. These include G->AT and G->in_degree for an undirected graph.

`LAGRAPH_MSG_LEN`

All LAGraph functions (except for *LAGraph_WallClockTime*) have a final msg parameter that is a pointer to a user-allocated string in which an algorithm-specific error message can be returned. If msg is NULL, no error message is returned. This is not itself an error condition, it just indicates that the caller does not need the message returned. If the message string is provided but no error occurs, an empty string is returned.

LAGRAPH_MSG_LEN is the minimum required length of a message string.

For example, the following call computes the breadth-first-search of an LAGraph_Graph G, starting at a given source node. It returns a status of zero if it succeeds and a negative value on failure.

```
GrB_Vector level, parent ;
char msg [LAGRAPH_MSG_LEN] ;
int status = LAGr_BreadthFirstSearch (&level, &parent, G, src, msg) ;
if (status < 0)
{
    printf ("status %d, error: %s\n", status, msg) ;
    ... take corrective action here ...
}
```

Error handling is simplified by the *LAGRAPH_TRY* / LAGRAPH_CATCH mechanism described below. For example, assuming the user application defines a single LAGRAPH_CATCH mechanism for all error handling, the above example would become:

```
GrB_Vector level, parent ;
char msg [LAGRAPH_MSG_LEN] ;
#define LAGRAPH_CATCH(status)                              \
{                                                          \
    printf ("status %d, error: %s\n", status, msg) ;       \
    ... take corrective action here ...                    \
}
...
LAGRAPH_TRY (LAGr_BreadthFirstSearch (&level, &parent, G, src, msg)) ;
```

The advantage of the second use case is that the error-handling block of code needs to be written only once.

`LAGRAPH_TRY`(LAGraph_method)

LAGRAPH_TRY: try an LAGraph method and check for errors.

In a robust application, the return values from each call to LAGraph and GraphBLAS should be checked, and corrective action should be taken if an error occurs. The LAGRAPH_TRY and *GRB_TRY* macros assist in this effort.

LAGraph and GraphBLAS are written in C, and so they cannot rely on the try/catch mechanism of C++. To accomplish a similar goal, each LAGraph file must `#define` its own file-specific macro called LAGRAPH_CATCH. The typical usage of macro is to free any temporary matrices/vectors or workspace when an error occurs, and then "throw" the error by returning to the caller. A user application may also `#define LAGRAPH_CATCH` and use these macros. The LAGRAPH_CATCH macro takes a single argument, which is the return value from an LAGraph method.

A typical example of a user function that calls LAGraph might define LAGRAPH_CATCH as follows. Suppose workvector is a GrB_vector used for computations internal to the mybfs function, and W is a (double *) space created by malloc.

```
// an example user-defined LAGRAPH_CATCH macro, which prints the error
// then frees any workspace or results, and returns to the caller:
#define LAGRAPH_CATCH(status)                                   \
{                                                               \
    printf ("LAGraph error: (%d): file: %s, line: %d\n%s\n",     \
        status, __FILE__, __LINE__, msg) ;                      \
    GrB_free (*parent) ;                                        \
    GrB_free (workvector) ;                                     \
    LAGraph_Free ((void **) &W, NULL) ;                         \
    return (status) ;                                           \
}

// an example user function that uses LAGRAPH_TRY / LAGRAPH_CATCH
int mybfs (LAGraph_Graph G, GrB_Vector *parent, int64_t src)
{
    GrB_Vector workvector = NULL ;
    double *W = NULL ;
    char msg [LAGRAPH_MSG_LEN] ;
    (*parent) = NULL ;
    LAGRAPH_TRY (LAGr_BreadthFirstSearch (NULL, parent, G, src, true,
        msg)) ;
    // ...
    return (GrB_SUCCESS) ;
}
```

LAGRAPH_TRY is defined as follows:

```
#define LAGRAPH_TRY(LAGraph_method)             \
{                                               \
    int LG_status = LAGraph_method ;            \
    if (LG_status < GrB_SUCCESS)                \
    {                                           \
        LAGRAPH_CATCH (LG_status) ;             \
    }                                           \
}
```

`GRB_TRY`(GrB_method)

GRB_TRY: LAGraph provides a similar functionality as *LAGRAPH_TRY* for calling GraphBLAS methods, with the GRB_TRY macro. GraphBLAS returns info = 0 (GrB_SUCCESS) or 1 (GrB_NO_VALUE) on success, and a value < 0 on failure. The user application must `#define GRB_CATCH` to use GRB_TRY.

GraphBLAS and LAGraph both use the convention that negative values are errors, and the LAGraph_status is a superset of the GrB_Info enum. As a result, the user can define LAGRAPH_CATCH and GRB_CATCH as the same operation. The main difference between the two would be the error message string. For LAGraph, the string is the last parameter, and LAGRAPH_CATCH can optionally print it out. For GraphBLAS, the GrB_error mechanism can return a string.

GRB_TRY is defined as follows:

```
#define GRB_TRY(GrB_method)                         \
{                                                   \
    GrB_Info LG_GrB_Info = GrB_method ;             \
    if (LG_GrB_Info < GrB_SUCCESS)                  \
    {                                               \
        GRB_CATCH (LG_GrB_Info) ;                   \
    }                                               \
}
```

CHAPTER

THREE

THE GRAPH OBJECT

The fundamental object in LAGraph is the `LAGraph_Graph`.

typedef struct *LAGraph_Graph_struct* ***LAGraph_Graph***

struct **`LAGraph_Graph_struct`**

LAGraph_Graph: a representation of a graph.

The LAGraph_Graph G contains a GrB_Matrix G->A as its primary component. For graphs represented with adjacency matrices, A(i,j) denotes the edge (i,j). Unlike GrB_* objects in GraphBLAS, the LAGraph_Graph data structure is not opaque. User applications have full access to its contents.

An LAGraph_Graph G contains two kinds of components:

a. Primary components of the graph, which fully define the graph.

b. Cached properties of the graph, which can be recreated any time.

Primary Components

GrB_Matrix **`A`**

the adjacency matrix of the graph

LAGraph_Kind **`kind`**

the kind of graph

Cached Properties

All of these components may be deleted or set to 'unknown' at any time. For example, if AT is NULL, then the transpose of A has not been computed. A scalar cached property of type LAGraph_Boolean would be set to LAGRAPH_UNKNOWN to denote that its value is unknown.

If present, the cached properties must be valid and accurate. If the graph changes, these cached properties can either be recomputed or deleted to denote the fact that they are unknown. This choice is up to individual LAGraph methods and utilities.

LAGraph methods can set non-scalar cached properties only if they are constructing the graph. They cannot modify them or create them if the graph is declared as a read-only object in the parameter list of the method.

GrB_Matrix **`AT`**

AT = A', the transpose of A, with the same type.

GrB_Vector **out_degree**

a GrB_INT64 vector of length m, if A is m-by-n. where out_degree(i) is the number of entries in A(i,:). If out_degree is sparse and the entry out_degree(i) is not present, then it is assumed to be zero.

GrB_Vector **in_degree**

a GrB_INT64 vector of length n, if A is m-by-n. where in_degree(j) is the number of entries in A(:,j). If in_degree is sparse and the entry in_degree(j) is not present, then it is assumed to be zero. If A is known to have a symmetric structure, the convention is that the degree is held in G->out_degree, and in G->in_degree is left as NULL.

LAGraph_Boolean **is_symmetric_structure**

For an undirected graph, this cached property will always be implicitly true and can be ignored. The matrix A for a directed weighted graph will typically be unsymmetric, but might have a symmetric structure. In that case, this scalar cached property can be set to true. By default, this cached property is set to LAGRAPH_UNKNOWN.

int64_t **nself_edges**

number of entries on the diagonal of A, or LAGRAPH_UNKNOWN if unknown. For the adjacency matrix of a directed or undirected graph, this is the number of self-edges in the graph.

GrB_Scalar **emin**

minimum edge weight: value, lower bound, or unknown

LAGraph_State **emin_state**

- VALUE: emin is equal to the smallest entry, min(G->A)
- BOUND: emin <= min(G->A)
- UNKNOWN: emin is unknown

GrB_Scalar **emax**

maximum edge weight: value, upper bound, or unknown

LAGraph_State **emax_state**

- VALUE: emax is equal to the largest entry, max(G->A)
- BOUND: emax >= max(G->A)
- UNKNOWN: emax is unknown

enum **LAGraph_Boolean**

LAGraph_Boolean: a boolean value (true or false) or unknown (-1).

Values:

enumerator **LAGraph_FALSE**

the Boolean value is known to be false.

enumerator **`LAGraph_TRUE`**

the Boolean value is known to be true.

enumerator **`LAGraph_BOOLEAN_UNKNOWN`**

Boolean value is unknown.

enum **`LAGraph_State`**

LAGraph_State describes the status of a cached property of a graph. If the cached property is computed in floating-point arithmetic, it may have been computed with roundoff error, but it may still be declared as “value” if the roundoff error is expected to be small, or if the cached property was computed as carefully as possible (to within reasonable roundoff error). The “bound” state indicates that the cached property is an upper or lower bound, depending on the particular cached property. If computed in floating-point arithmetic, an “upper bound” cached property may be actually slightly lower than the actual upper bound, because of floating-point roundoff.

Values:

enumerator **`LAGraph_VALUE`**

cached property is a known value.

enumerator **`LAGraph_BOUND`**

cached property is a bound. The bound is upper or lower, depending on the particular cached property.

enumerator **`LAGraph_STATE_UNKNOWN`**

the property is unknown.

enum **`LAGraph_Kind`**

LAGraph_Kind: the kind of a graph. Currently, only two types of graphs are supported: undirected graphs and directed graphs. Edge weights are assumed to be present. Unweighted graphs can be represented by setting all entries present in the sparsity structure to the same value, typically 1. Additional types of graphs will be added in the future.

Values:

enumerator **`LAGraph_ADJACENCY_UNDIRECTED`**

undirected graph. G->A is square and symmetric; both upper and lower triangular parts are present. A(i,j) is the edge (i,j). Results are undefined if A is unsymmetric.

enumerator **`LAGraph_ADJACENCY_DIRECTED`**

G->A is square; A(i,j) is the edge (i,j).

directed graph.

enumerator **`LAGraph_KIND_UNKNOWN`**

unknown kind of graph (-1).

3.1 Basic Graph Functions

int **LAGraph_New**(*LAGraph_Graph* *G, GrB_Matrix *A, *LAGraph_Kind* kind, char *msg)

LAGraph_New: creates a new graph G. The cached properties G->AT, G->out_degree, and G->in_degree are set to NULL, and scalar cached properties are set to LAGRAPH_UNKNOWN.

Parameters

- **G** – **[out]** handle to the newly created graph, as &G.
- **A** – **[inout]** adjacency matrix. A is moved into G as G->A, and A itself is set to NULL to denote that is now a part of G. That is, { G->A = A ; A = NULL ; } is performed. When G is deleted, G->A is freed. If A is NULL, the graph is invalid until G->A is set.
- **kind** – **[in]** the kind of graph to create. This may be LAGRAPH_UNKNOWN, which must then be revised later before the graph can be used.
- **msg** – **[inout]** any error messages.

Return values

- **GrB_SUCCESS** – if successful.
- **GrB_NULL_POINTER** – if G is null.

Returns

any GraphBLAS errors that may have been encountered.

int **LAGraph_Delete**(*LAGraph_Graph* *G, char *msg)

LAGraph_Delete: frees a graph G. The adjacency matrix G->A and the cached properties G->AT, G->out_degree, and G->in_degree are all freed.

Parameters

- **G** – **[inout]** handle to the graph to be free. *G is NULL on output. To keep G->A while deleting the graph G, use: { A = G->A ; G->A = NULL ; LAGraph_Delete (&G, msg) ; }
- **msg** – **[inout]** any error messages.

Return values

GrB_SUCCESS – if successful.

Returns

any GraphBLAS errors that may have been encountered.

int **LAGraph_DeleteCached**(*LAGraph_Graph* G, char *msg)

LAGraph_DeleteCached: frees all cached properies of a graph G. The graph is still valid. This method should be used if G->A changes, since such changes will normally invalidate G->AT, G->out_degree, and/or G->in_degree.

Parameters

- **G** – **[inout]** handle to the graph to modified. The graph G remains valid on output, but with all cached properties freed. G may be NULL, in which case nothing is done.
- **msg** – **[inout]** any error messages.

Return values

GrB_SUCCESS – if successful.

Returns

any GraphBLAS errors that may have been encountered.

int **LAGraph_Cached_AT**(*LAGraph_Graph* G, char *msg)

LAGraph_Cached_AT: constructs G->AT, the transpose of G->A. This matrix is required by some of the algorithms. Basic algorithms may construct G->AT if they require it. The matrix G->AT is then available for subsequent use. If G->A changes, G->AT should be freed and recomputed. If G->AT already exists, it is left unchanged (even if it is not equal to the transpose of G->A). As a result, if G->A changes, G->AT should be explictly freed.

Parameters

- `G` – **[inout]** graph for which G->AT is computed.
- `msg` – **[inout]** any error messages.

Return values

- `GrB_SUCCESS` – if successful.
- `GrB_NULL_POINTER` – if G is NULL.
- `LAGRAPH_CACHE_NOT_NEEDED` – if G->kind is LAGraph_ADJACENCY_UNDIRECTED.
- `LAGRAPH_INVALID_GRAPH` – if G is invalid (G->A missing, or G->kind not a recognized kind).

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Cached_IsSymmetricStructure`**(*LAGraph_Graph* G, char *msg)

LAGraph_Cached_IsSymmetricStructure: determines if the sparsity structure of G->A is symmetric (ignoring its values). If G->kind denotes that the graph is undirected, this cached property is implicitly true (and not checked). Otherwise, this method determines if the structure of G->A for a directed graph G has a symmetric sparsity structure. No work is performed if the cached property is already known.

Parameters

- `G` – **[inout]** graph for which G->is_symmetric_structure is computed.
- `msg` – **[inout]** any error messages.

Return values

- `GrB_SUCCESS` – if successful.
- `GrB_NULL_POINTER` – if G is NULL.
- `LAGRAPH_INVALID_GRAPH` – if G is invalid (G->A missing, or G->kind not a recognized kind).

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Cached_OutDegree`**(*LAGraph_Graph* G, char *msg)

LAGraph_Cached_OutDegree: computes G->out_degree. No work is performed if it already exists in G.

Parameters

- `G` – **[inout]** graph for which G->out_degree is computed.
- `msg` – **[inout]** any error messages.

Return values

- `GrB_SUCCESS` – if successful.
- `GrB_NULL_POINTER` – if G is NULL.
- `LAGRAPH_INVALID_GRAPH` – if G is invalid (G->A missing, or G->kind not a recognized kind).

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Cached_InDegree`**(*LAGraph_Graph* G, char *msg)

LAGraph_Cached_InDegree computes G->in_degree. No work is performed if it already exists in G. If G is undirected, G->in_degree is never computed and remains NULL (the method returns LAGRAPH_CACHE_NOT_NEEDED). No work is performed if it is already exists in G.

Performance note: for SuiteSparse:GraphBLAS, if G->A is held by row (the default format), then computing G->in_degree is fastest if G->AT is known. If G->AT will be needed anyway, compute it first with LAGraph_Cached_AT, and then call LAGraph_Cached_Indegree. This is optional; if G->AT is not known, then G->in_degree is computed from G->A instead.

Parameters

- **`G`** – **[inout]** graph for which G->in_degree is computed.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if G is NULL.
- **`LAGRAPH_CACHE_NOT_NEEDED`** – if G->kind is LAGraph_ADJACENCY_UNDIRECTED.
- **`LAGRAPH_INVALID_GRAPH`** – if G is invalid (G->A missing, or G->kind not a recognized kind).

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Cached_NSelfEdges`**(*LAGraph_Graph* G, char *msg)

LAGraph_Cached_NSelfEdges: computes G->nself_edges, the number of diagonal entries that appear in the G->A matrix. For an undirected or directed graph with an adjacency matrix G->A, these are the number of self-edges in G. No work is performed it is already computed.

Parameters

- **`G`** – **[inout]** graph for which G->nself_edges is computed.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if G is NULL.
- **`LAGRAPH_INVALID_GRAPH`** – if G is invalid (G->A missing, or G->kind not a recognized kind).

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Cached_EMin`**(*LAGraph_Graph* G, char *msg)

LAGraph_Cached_EMin: computes the G->emin, the smallest entry in G->A. Not computed if G->emin already exists.

Parameters

- **`G`** – **[inout]** graph for which G->emin is computed.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.

- **`GrB_NOT_IMPLEMENTED`** – if G does not have a built-in real type: GrB_(BOOL, INT8, INT16, INT32, INT64, UINT8, UINT16, UINT32, UINT64, FP32, OR FP64).
- **`GrB_NULL_POINTER`** – if G is NULL.
- **`LAGRAPH_INVALID_GRAPH`** – if G is invalid (G->A missing, or G->kind not a recognized kind).

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Cached_EMax`**(*LAGraph_Graph* G, char *msg)

LAGraph_Cached_EMax: computes the G->emax, the largest entry in G->A. Not computed if G->emax already exists.

Parameters

- **`G`** – **[inout]** graph for which G->emax is computed.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NOT_IMPLEMENTED`** – if G does not have a built-in real type: GrB_(BOOL, INT8, INT16, INT32, INT64, UINT8, UINT16, UINT32, UINT64, FP32, OR FP64).
- **`GrB_NULL_POINTER`** – if G is NULL.
- **`LAGRAPH_INVALID_GRAPH`** – if G is invalid (G->A missing, or G->kind not a recognized kind).

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_DeleteSelfEdges`**(*LAGraph_Graph* G, char *msg)

LAGraph_DeleteSelfEdges: removes any diagonal entries from G->A. Most cached properties are cleared or set to LAGRAPH_UNKNOWN. G->nself_edges is set to zero, and G->is_symmetric_structure is left unchanged.

Parameters

- **`G`** – **[inout]** graph for which G->A is modified.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if G is NULL.
- **`LAGRAPH_INVALID_GRAPH`** – if G is invalid (G->A missing, or G->kind not a recognized kind).

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_CheckGraph`**(*LAGraph_Graph* G, char *msg)

LAGraph_CheckGraph: determines if a graph is valid. Only basic checks are performed on the cached properties, taking O(1) time.

Parameters

- **`G`** – **[in]** graph to check.

- `msg` – **[inout]** any error messages.

Return values

- `GrB_SUCCESS` – if successful.
- `GrB_NULL_POINTER` – if G is NULL.
- `LAGRAPH_INVALID_GRAPH` – if G is invalid: G->A missing, G->kind not a recognized kind, G->AT present but has the wrong dimensions or its type does not match G->A, G->in_degree/out_degree present but with the wrong dimension or type (in/out_degree must be GrB_INT64).

Returns

any GraphBLAS errors that may have been encountered.

CHAPTER

FOUR

ALGORITHMS

Algorithms come in two flavors: *Basic* and *Advanced*.

4.1 Basic

Basic algorithm are meant to be easy to use. A single basic algorithm may encompass many underlying Advanced algorithms, each with various parameters that may be controlled. For the Basic API, these parameters are determined automatically. Cached graph properties may be determined, and as a result, the graph G is both an input and an output of these methods, since they may be modified.

LAGraph Basic algorithms are named with the `LAGraph_*` prefix.

int **`LAGraph_TriangleCount`**(uint64_t *ntriangles, *LAGraph_Graph* G, char *msg)

LAGraph_TriangleCount: count the triangles in a graph. This is a Basic algorithm (G->nself_edges, G->out_degree, G->is_symmetric_structure are computed, if not present).

Parameters

- **`ntriangles`** – **[out]** the number of triangles in G.
- **`G`** – **[inout]** the graph, which must by undirected, or directed but with a symmetric structure. No self loops can be present.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if G or ntriangles are NULL.
- **`LAGRAPH_INVALID_GRAPH`** – if G is invalid (*LAGraph_CheckGraph* failed).
- **`LAGRAPH_NO_SELF_EDGES_ALLOWED`** – if G has any self-edges.
- **`LAGRAPH_SYMMETRIC_STRUCTURE_REQUIRED`** – if G is directed with an unsymmetric G->A matrix.

Returns

any GraphBLAS errors that may have been encountered.

4.2 Advanced

The Advanced algorithms require the caller to select the algorithm and choose any parameter settings. G is not modified, and so it is an input-only parameter to these methods. If an Advanced algorithm requires a cached graph property to be computed, it must be computed prior to calling the Advanced method.

Advanced algorithms are named with the `LAGr_*` prefix, to distinguish them from Basic algorithms.

int **LAGr_SortByDegree**(int64_t **P, const *LAGraph_Graph* G, bool byout, bool ascending, char *msg)

LAGr_SortByDegree sorts the nodes of a graph by their out or in degrees. The graph G->A itself is not changed. Refer to LAGr_TriangleCount for an example of how to permute G->A after calling this function. The output &P must be freed by LAGraph_Free. This method requires G->out_degree or G->in_degree to already be computed.

Parameters

- `P` – **[out]** permutation of the integers 0..n-1.
- `G` – **[in]** graph of n nodes.
- `byout` – **[in]** if true, sort by out-degree, else sort by in-degree.
- `ascending` – **[in]** if true, sort in ascending order, else descending.
- `msg` – **[inout]** any error messages.

Return values

- `GrB_NULL_POINTER` – if P or G are NULL.
- `LAGRAPH_NOT_CACHED` – if G->in_degree or G->out_degree is not computed (whichever one is required).
- `LAGRAPH_INVALID_GRAPH` – if G is invalid (*LAGraph_CheckGraph* failed).

Returns

any GraphBLAS errors that may have been encountered.

int **LAGr_SampleDegree**(double *sample_mean, double *sample_median, const *LAGraph_Graph* G, bool byout, int64_t nsamples, uint64_t seed, char *msg)

LAGr_SampleDegree computes an estimate of the median and mean of the out or in degree, by randomly sampling the G->out_degree or G->in_degree vector. This method requires G->out_degree or G->in_degree to already be computed.

Parameters

- `sample_mean` – **[out]** sampled mean of the degree.
- `sample_median` – **[out]** sampled median of the degree.
- `G` – **[in]** graph to sample.
- `byout` – **[in]** if true, sample out-degree, else sample in-degree.
- `nsamples` – **[in]** number of samples to take.
- `seed` – **[in]** random number seed.
- `msg` – **[inout]** any error messages.

Return values

- `GrB_NULL_POINTER` – if sample_mean, sample_median, or G are NULL.
- `LAGRAPH_NOT_CACHED` – if G->in_degree or G->out_degree is not computed (whichever one is required).
- `LAGRAPH_INVALID_GRAPH` – if G is invalid (*LAGraph_CheckGraph* failed).

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGr_BreadthFirstSearch`**(GrB_Vector *level, GrB_Vector *parent, const *LAGraph_Graph* G, GrB_Index src, char *msg)

LAGr_BreadthFirstSearch: breadth-first search of a graph, computing the breadth-first-search tree and/or the level of the nodes encountered. This is an Advanced algorithm. G->AT and G->out_degree are required to use the fastest push/pull method when using SuiteSparse:GraphBLAS. If these cached properties are not present, or if a vanilla GraphBLAS library is being used, then a push-only method is used (which can be slower). G is not modified; that is, G->AT and G->out_degree are not computed if not already cached.

Parameters

- **`level`** – **[out]** If non-NULL on input, on successful return, it contains the levels of each node reached. The src node is assigned level 0. If a node i is not reached, level(i) is not present. The level vector is not computed if NULL.
- **`parent`** – **[out]** If non-NULL on input, on successful return, it contains parent node IDs for each node reached, where parent(i) is the node ID of the parent of node i. The src node will have itself as its parent. If a node i is not reached, parent(i) is not present. The parent vector is not computed if NULL.
- **`G`** – **[in]** The graph, directed or undirected.
- **`src`** – **[in]** The index of the src node (0-based)
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_INVALID_INDEX`** – if src is invalid.
- **`GrB_NULL_POINTER`** – if both level and parent are NULL, or if G is NULL.
- **`LAGRAPH_INVALID_GRAPH`** – Graph is invalid (*LAGraph_CheckGraph* failed).

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGr_ConnectedComponents`**(GrB_Vector *component, const *LAGraph_Graph* G, char *msg)

LAGr_ConnectedComponents: connected components of an undirected graph. This is an Advanced algorithm (G->is_symmetric_structure must be known).

Parameters

- **`component`** – **[out]** component(i)=s if node i is in the component whose representative node is s. If node i has no edges, it is placed in its own component, and thus the component vector is always dense.
- **`G`** – **[in]** input graph to find the components for. The graph must be undirected, or G->is_symmetric_structure must be true.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if G or component are NULL.
- **`LAGRAPH_INVALID_GRAPH`** – Graph is invalid (*LAGraph_CheckGraph* failed).
- **`LAGRAPH_SYMMETRIC_STRUCTURE_REQUIRED`** – if G is directed with an unsymmetric G->A matrix.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGr_SingleSourceShortestPath`**(GrB_Vector *path_length, const *LAGraph_Graph* G, GrB_Index src, GrB_Scalar Delta, char *msg)

LAGr_SingleSourceShortestPath: single-source shortest paths. This is an Advanced algorithm (G->emin is required for best performance). The graph G must have an adjacency matrix of type GrB_INT32, GrB_INT64, GrB_UINT32, GrB_UINT64, GrB_FP32, or GrB_FP64. If G->A has any other type, GrB_NOT_IMPLEMENTED is returned.

Parameters

- **`path_length`** – **[out]** path_length (i) is the length of the shortest path from the source node to node i. The path_length vector is dense. If node (i) is not reachable from the src node, then path_length (i) is set to INFINITY for GrB_FP32 and FP32, or the maximum integer for GrB_INT32, INT64, UINT32, or UINT64.
- **`G`** – **[in]** input graph.
- **`src`** – **[in]** source node.
- **`Delta`** – **[in]** for delta stepping.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if G or path_length are NULL.
- **`GrB_INVALID_INDEX`** – if src is invalid.
- **`GrB_EMPTY_OBJECT`** – if Delta does not contain a value.
- **`GrB_NOT_IMPLEMENTED`** – if the type is not supported.
- **`LAGRAPH_INVALID_GRAPH`** – Graph is invalid (*LAGraph_CheckGraph* failed).

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGr_Betweenness`**(GrB_Vector *centrality, const *LAGraph_Graph* G, const GrB_Index *sources, int32_t ns, char *msg)

LAGr_Betweenness: betweeness centrality metric. This methods computes an approximation of the betweeness-centrality metric of all nodes in the graph. Only a few given source nodes are used for the approximation. This is an Advanced algorithm (G->AT is required).

Parameters

- **`centrality`** – **[out]** centrality(i) is the metric for node i.
- **`G`** – **[in]** input graph.
- **`sources`** – **[in]** source vertices to compute shortest paths, size ns
- **`ns`** – **[in]** number of source vertices.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if G, centrality, and/our sources are NULL.

- **GrB_INVALID_INDEX** – if any source node is invalid.
- **LAGRAPH_INVALID_GRAPH** – Graph is invalid (*LAGraph_CheckGraph* failed).
- **LAGRAPH_NOT_CACHED** – if G->AT is required but not present.

Returns

any GraphBLAS errors that may have been encountered.

int **LAGr_PageRank**(GrB_Vector *centrality, int *iters, const *LAGraph_Graph* G, float damping, float tol, int itermax, char *msg)

LAGr_PageRank: computes the standard PageRank of a directed graph G. Sinks (nodes with no out-going edges) are properly handled. This method should be used for production, not for the GAP benchmark. This is an Advanced algorithm (G->AT and G->out_degree are required).

Parameters

- **centrality** – **[out]** centrality(i) is the PageRank of node i.
- **iters** – **[out]** number of iterations taken.
- **G** – **[in]** input graph.
- **damping** – **[in]** damping factor (typically 0.85).
- **tol** – **[in]** stopping tolerance (typically 1e-4).
- **itermax** – **[in]** maximum number of iterations (typically 100).
- **msg** – **[inout]** any error messages.

Return values

- **GrB_SUCCESS** – if successful.
- **GrB_NULL_POINTER** – if G, centrality, and/our iters are NULL.
- **LAGRAPH_NOT_CACHED** – if G->AT is required but not present, or if G->out_degree is not present.
- **LAGRAPH_INVALID_GRAPH** – Graph is invalid (*LAGraph_CheckGraph* failed).

Returns

any GraphBLAS errors that may have been encountered.

int **LAGr_TriangleCount**(uint64_t *ntriangles, const *LAGraph_Graph* G, *LAGr_TriangleCount_Method* *method, *LAGr_TriangleCount_Presort* *presort, char *msg)

LAGr_TriangleCount: count the triangles in a graph (advanced API).

Parameters

- **ntriangles** – **[out]** the number of triangles in G.
- **G** – **[in]** The graph, which must be undirected or have G->is_symmetric_structure true, with no self loops. G->nself_edges, G->out_degree, and G->is_symmetric_structure are required.
- **method** – **[inout]** specifies which algorithm to use, and returns the method chosen. If NULL, the AutoMethod is used, and the method is not reported. Also see the LAGr_TriangleCount_Method enum description.
- **presort** – **[inout]** controls the presort of the graph, and returns the presort chosen. If NULL, the AutoSort is used, and the presort method is not reported. Also see the description of the LAGr_TriangleCount_Presort enum.
- **msg** – **[inout]** any error messages.

Return values

- `GrB_SUCCESS` – if successful.
- `GrB_NULL_POINTER` – if G or ntriangles are NULL.
- `LAGRAPH_INVALID_GRAPH` – Graph is invalid (*LAGraph_CheckGraph* failed).
- `LAGRAPH_NO_SELF_EDGES_ALLOWED` – if G has any self-edges, or if G->nself_edges is not computed.
- `LAGRAPH_SYMMETRIC_STRUCTURE_REQUIRED` – if G is directed with an unsymmetric G->A matrix.
- `LAGRAPH_NOT_CACHED` – if G->out_degree is not present in G.
- `GrB_INVALID_VALUE` – method or presort are invalid.

Returns

any GraphBLAS errors that may have been encountered.

enum **`LAGr_TriangleCount_Method`**

LAGr_TriangleCount_Method: an enum to select the method used to count the number of triangles.

Values:

enumerator **`LAGr_TriangleCount_AutoMethod`**

auto selection of method

enumerator **`LAGr_TriangleCount_Burkhardt`**

sum (sum ((A^2) .* A)) / 6

enumerator **`LAGr_TriangleCount_Cohen`**

sum (sum ((L * U) .* A)) / 2

enumerator **`LAGr_TriangleCount_Sandia_LL`**

sum (sum ((L * L) .* L))

enumerator **`LAGr_TriangleCount_Sandia_UU`**

sum (sum ((U * U) .* U))

enumerator **`LAGr_TriangleCount_Sandia_LUT`**

sum (sum ((L * U’) .* L))

enumerator **`LAGr_TriangleCount_Sandia_ULT`**

sum (sum ((U * L’) .* U))

enum **`LAGr_TriangleCount_Presort`**

LAGr_TriangleCount_Presort: an enum to control if/how the matrix is sorted prior to counting triangles.

Values:

enumerator **`LAGr_TriangleCount_NoSort`**

no sort

enumerator **`LAGr_TriangleCount_Ascending`**

sort by degree, ascending.

enumerator **`LAGr_TriangleCount_Descending`**

sort by degree, descending.

enumerator **`LAGr_TriangleCount_AutoSort`**

auto selection of presort: No presort is done for the Burkhardt or Cohen methods, and no sort is done for the Sandia_* methods if the sampled mean out-degree is <= 4 * the sample median out-degree. Otherwise: sort in ascending order for Sandia_LL and Sandia_LUT, descending ordering for Sandia_UU and Sandia_ULT. On output, presort is modified to reflect the sorting method used (NoSort, Ascending, or Descending).

CHAPTER

FIVE

UTILITY FUNCTIONS

5.1 Input/Output Functions

int **LAGraph_MMRead**(GrB_Matrix *A, FILE *f, char *msg)

LAGraph_MMRead: reads a matrix in MatrixMarket format. The file format used here is compatible with all variations of the Matrix Market "coordinate" and "array" format (http://www.nist.gov/MatrixMarket), for sparse and dense matrices respectively. The format is fully described in LAGraph/papers/MatrixMarket.pdf, and summarized here (with extensions for LAGraph).

5.1.1 First Line

The first line of the file starts with `%%MatrixMarket`, with the following format:

```
%%MatrixMarket matrix <fmt> <type> <storage>
```

<fmt>

One of:

- coordinate : sparse matrix in triplet form
- array : dense matrix in column-major form

Both formats are returned as a GrB_Matrix.

If not present, defaults to coordinate.

<type>

One of:

- real : returns as GrB_FP64
- integer : returns as GrB_INT64
- pattern : returns as GrB_BOOL
- complex : *currently not supported*

The return type can be modified by the `%%GraphBLAS` structured comment described below.

If not present, defaults to real.

<storage>

One of:

- **general**
  the matrix has no symmetry properties (or at least none that were exploited when the file was created).
- **Hermitian**
  square complex matrix with A(i,j) = conj (A(j,i)). All entries on the diagonal are real. Each off-diagonal entry in the file creates two entries in the GrB_Matrix that is returned.
- **symmetric**
  A(i,j) == A(j,i). Only entries on or below the diagonal appear in the file. Each off-diagonal entry in the file creates two entries in the GrB_Matrix that is returned.
- **skew-symmetric**
  A(i,j) == -A(i,j). There are no entries on the diagonal. Only entries below the diagonal appear in the file. Each off-diagonal entry in the file creates two entries in the GrB_Matrix that is returned.

The Matrix Market format is case-insensitive, so “hermitian” and “Hermitian” are treated the same.

If not present, defaults to general.

Not all combinations are permitted. Only the following are meaningful:

(1) (coordinate or array) x (real, integer, or complex) x (general, symmetric, or skew-symmetric)

(2) (coordinate or array) x (complex) x (Hermitian)

(3) (coodinate) x (pattern) x (general or symmetric)

5.1.2 Second Line

The second line is an optional extension to the Matrix Market format:

```
%%GraphBLAS type <entrytype>
```

<entrytype>
One of the 11 built-in types (bool, int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t, uint64_t, float, or double.

If this second line is included, it overrides the default GraphBLAS types for the Matrix Market <type> on line one of the file: real, pattern, and integer. The Matrix Market complex <type> is not yet supported.

5.1.3 Other Lines

Any other lines starting with “%” are treated as comments, and are ignored. Comments may be interspersed throughout the file. Blank lines are ignored. The Matrix Market header is optional in this routine (it is not optional in the Matrix Market format). The remaining lines are space delimited, and free format (one or more spaces can appear, and each field has arbitrary width).

5.1.4 Coordinate Format

For coordinate format, the first non-comment line must appear, and it must contain three integers:

```
nrows ncols nvals
```

For example, a 5-by-12 matrix with 42 entries would have:

```
5 12 42
```

Each of the remaining lines defines one entry. The order is arbitrary. If the Matrix Market <type> is real or integer, each line contains three numbers: row index, column index, and value. For example, if A(3,4) is equal to 5.77, a line:

```
3 4 5.77
```

would appear in the file. The indices in the Matrix Market are 1-based, so this entry becomes A(2,3) in the GrB_Matrix returned to the caller. If the <type> is pattern, then only the row and column index appears. If <type> is complex, four values appear. If A(8,4) has a real part of 6.2 and an imaginary part of -9.3, then the line is:

```
8 4 6.2 -9.3
```

and since the file is 1-based but a GraphBLAS matrix is always 0-based, one is subtracted from the row and column indices in the file, so this entry becomes A(7,3). Note however that LAGraph does not yet support complex types.

5.1.5 Array Format

For array format, the first non-comment line must appear, and it must contain just two integers:

```
nrows ncols
```

A 5-by-12 matrix would have this as the first non-comment line after the header:

```
5 12
```

Each of the remaining lines defines one entry, in column major order. If the <type> is real or integer, this is the value of the entry. An entry if <type> of complex consists of two values, the real and imaginary part (not yet supported). The <type> cannot be pattern in this case.

5.1.6 Infinity & Not-A-Number

For both coordinate and array formats, real and complex values may use the terms `INF`, `+INF`, `-INF`, and `NAN` to represent floating-point infinity and NaN values, in either upper or lower case.

According to the Matrix Market format, entries are always listed in column-major order. This rule is follwed by *LAGraph_MMWrite* . However, LAGraph_MMRead can read the entries in any order.

Parameters

- **`A`** – **[out]** handle of the matrix to create.
- **`f`** – **[inout]** handle to an open file to read from.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if A or f are NULL.
- **`LAGRAPH_IO_ERROR`** – if the file could not be read or contains a matrix with an invalid Matrix Market format.
- **`GrB_NOT_IMPLEMENTED`** – if the type is not supported. Complex types (GxB_FC32 and GxB_FC64 in SuiteSparse:GraphBLAS) are not yet supported.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_MMWrite`**(GrB_Matrix A, FILE *f, FILE *fcomments, char *msg)

LAGraph_MMWrite: writes a matrix in MatrixMarket format. Refer to *LAGraph_MMRead* for a description of the output file format. The MatrixMarket header line always appears, followed by the second line containing the GraphBLAS type:

```
%%GraphBLAS type <entrytype>
```

Parameters

- **`A`** – **[in]** matrix to write.
- **`f`** – **[inout]** handle to an open file to write to.
- **`fcomments`** – **[in]** optional handle to an open file containing comments; may be NULL.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if A or f are NULL.
- **`LAGRAPH_IO_ERROR`** – if the file could not be written to.
- **`GrB_NOT_IMPLEMENTED`** – if the type is not supported. Complex types (GxB_FC32 and GxB_FC64 in SuiteSparse:GraphBLAS) are not yet supported.

Returns

any GraphBLAS errors that may have been encountered.

double **`LAGraph_WallClockTime`**(void)

LAGraph_WallClockTime returns the current wall clock time. Normally, this is simply a wrapper for omp_get_wtime, if OpenMP is in use. Otherwise, an OS-specific timing function is called. Note that unlike all other LAGraph functions, this function does not return an error condition, nor does it have a msg string parameter. Instead, it returns the current wall clock time (in seconds) since some fixed point in the past.

Returns

the current wall clock time.

5.2 Matrix Structure Functions

int **`LAGraph_Matrix_Structure`**(GrB_Matrix *C, GrB_Matrix A, char *msg)

LAGraph_Matrix_Structure: returns the sparsity structure of a matrix A as a boolean (GrB_BOOL) matrix C. If A(i,j) appears in the sparsity structure of A, then C(i,j) is set to true. The sparsity structure of A and C are identical.

Parameters

- **`C`** – **[out]** A boolean matrix with same structure of A, with C(i,j) true if A(i,j) appears in the sparsity structure of A.
- **`A`** – **[in]** matrix to compute the structure for.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if A or C are NULL.

Returns
any GraphBLAS errors that may have been encountered.

int **`LAGraph_Vector_Structure`**(GrB_Vector *w, GrB_Vector u, char *msg)

LAGraph_Vector_Structure: returns the sparsity structure of a vector u as a boolean (GrB_BOOL) vector w. If u(i) appears in the sparsity structure of u, then w(i) is set to true. The sparsity structure of u and w are identical.

Parameters

- **`w`** – **[out]** A boolean vector with same structure of u, with w(i) true if u(i,j) appears in the sparsity structure of u.
- **`u`** – **[in]** vector to compute the structure for.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if w or u are NULL.

Returns
any GraphBLAS errors that may have been encountered.

5.3 Matrix Comparison Functions

int **`LAGraph_Matrix_IsEqual`**(bool *result, const GrB_Matrix A, const GrB_Matrix B, char *msg)

LAGraph_Matrix_IsEqual compares two matrices for exact equality. If the two matrices have different data types, the result is always false (no typecasting is performed). Only the 11 built-in GrB* types are supported. If both A and B are NULL, the return value is true. If A and/or B are floating-point types and contain NaN's, result is false.

Parameters

- **`result`** – **[out]** true if A and B are exactly equal, false otherwise.
- **`A`** – **[in]** matrix to compare.
- **`B`** – **[in]** matrix to compare.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if result is NULL.
- **`GrB_NOT_IMPLEMENTED`** – if A or B has a user-defined type.

Returns
any GraphBLAS errors that may have been encountered.

int **`LAGraph_Matrix_IsEqualOp`**(bool *result, const GrB_Matrix A, const GrB_Matrix B, const GrB_BinaryOp op, char *msg)

LAGraph_Matrix_IsEqualOp compares two matrices using the given binary operator. The op may be built-in or user-defined. The two matrices may have different types and still be determined to be equal. To be equal, two matrices must have the same sparsity structure, and op(aij,bij) must return true for all pairs of entries aij and bij that appear in the structure of both A and B. The matrices A and/or B can have any type, as long as they are valid inputs to the op. If both A and B are NULL, the return value is true.

Parameters

- **`result`** – **[out]** true if A and B are equal (per the op), false otherwise.
- **`A`** – **[in]** matrix to compare.
- **`B`** – **[in]** matrix to compare.
- **`op`** – **[in]** operator for the comparison.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if result or op are NULL.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Vector_IsEqual`**(bool *result, const GrB_Vector u, const GrB_Vector v, char *msg)

LAGraph_Vector_IsEqual compares two vectors for exact equality. If the two vectors have different data types, the result is always false (no typecasting is performed). Only the 11 built-in GrB* types are supported. If both u and v are NULL, the return value is true. If u and/or v are floating-point types and contain NaN's, result is false.

Parameters

- **`result`** – **[out]** true if u and v are exactly equal, false otherwise.
- **`u`** – **[in]** vector to compare.
- **`v`** – **[in]** vector to compare.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if result is NULL.
- **`GrB_NOT_IMPLEMENTED`** – if u or v has a user-defined type.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Vector_IsEqualOp`**(bool *result, const GrB_Vector u, const GrB_Vector v, const GrB_BinaryOp op, char *msg)

LAGraph_Vector_IsEqualOp compares two vectors using the given binary operator. The op may be built-in or user-defined. The two vectors may have different types and still be determined to be equal. To be equal, two vectors must have the same sparsity structure, and op(ui,vi) must return true for all pairs of entries ui and vi that appear in the structure of both u and v. The vectors u and/or v can have any type, as long as they are valid inputs to the op. If both u and v are NULL, the return value is true.

Parameters

- **`result`** – **[out]** true if u and v are equal (per the op), false otherwise.
- **`u`** – **[in]** vector to compare.
- **`v`** – **[in]** vector to compare.
- **`op`** – **[in]** operator for the comparison.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if result or op are NULL.

Returns

any GraphBLAS errors that may have been encountered.

5.4 Introspecting Types

int **`LAGraph_NameOfType`**(char *name, GrB_Type type, char *msg)

LAGraph_NameOfType returns the name of a GraphBLAS type as a string. The names for the 11 built-in types (GrB_BOOL, GrB_INT8, etc) correspond to the names of the corresponding C types (bool, int8_t, etc).

Parameters

- **`name`** – **[out]** name of the type: user provided array of size at least LAGRAPH_MAX_NAME_LEN.
- **`type`** – **[in]** GraphBLAS type to find the name of.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if name or type are NULL.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_TypeFromName`**(GrB_Type *type, char *name, char *msg)

LAGraph_TypeFromName: returns the GrB_Type corresponding to its name. That is, given the string "bool", this method returns GrB_BOOL.

Parameters

- **`type`** – **[out]** GraphBLAS type corresponding to the given name string.
- **`name`** – **[in]** name of the type: a null-terminated string.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if name or type are NULL.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_SizeOfType`**(size_t *size, GrB_Type type, char *msg)

LAGraph_SizeOfType: returns sizeof(…) of a GraphBLAS GrB_Type. For example, if given the GrB_Type of GrB_FP64, the value sizeof(double) is returned.

Parameters

- **`size`** – **[out]** size of the type
- **`type`** – **[in]** GraphBLAS type to find the size of.
- **`msg`** – **[inout]** any error messages.

Return values

- `GrB_SUCCESS` – if successful.
- `GrB_NULL_POINTER` – if size or type are NULL.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Matrix_TypeName`**(char *name, GrB_Matrix A, char *msg)

LAGraph_Matrix_TypeName: returns the name of the GrB_Type of a GrB_Matrix.

Parameters

- **`name`** – **[out]** name of the type of the matrix: user provided array of size at least LAGRAPH_MAX_NAME_LEN.
- **`A`** – **[in]** GraphBLAS matrix to find the type name of.
- **`msg`** – **[inout]** any error messages.

Return values

- `GrB_SUCCESS` – if successful.
- `GrB_NULL_POINTER` – if name or A are NULL.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Vector_TypeName`**(char *name, GrB_Vector v, char *msg)

LAGraph_Vector_TypeName: returns the name of the GrB_Type of a GrB_Vector.

Parameters

- **`name`** – **[out]** name of the type of the vector: user provided array of size at least LAGRAPH_MAX_NAME_LEN.
- **`v`** – **[in]** GraphBLAS vector to find the type name of.
- **`msg`** – **[inout]** any error messages.

Return values

- `GrB_SUCCESS` – if successful.
- `GrB_NULL_POINTER` – if name or v are NULL.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Scalar_TypeName`**(char *name, GrB_Scalar s, char *msg)

LAGraph_Scalar_TypeName: returns the name of the GrB_Type of a GrB_Scalar.

Parameters

- **`name`** – **[out]** name of the type of the scalar: user provided array of size at least LAGRAPH_MAX_NAME_LEN.
- **`s`** – **[in]** GraphBLAS scalar to find the type name of.
- **`msg`** – **[inout]** any error messages.

Return values

- `GrB_SUCCESS` – if successful.
- `GrB_NULL_POINTER` – if name or s are NULL.

Returns
any GraphBLAS errors that may have been encountered.

5.5 Printing

int **`LAGraph_Graph_Print`**(const *LAGraph_Graph* G, *LAGraph_PrintLevel* pr, FILE *f, char *msg)

LAGraph_Graph_Print: prints the contents of a graph to a file in a human- readable form. This method is not meant for saving a graph to a file; see *LAGraph_MMWrite* for that method.

Parameters

- **`G`** – **[in]** graph to display.
- **`pr`** – **[in]** print level.
- **`f`** – **[inout]** handle to an open file to write to.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if G or f are NULL.
- **`LAGRAPH_INVALID_GRAPH`** – if G is invalid (*LAGraph_CheckGraph* failed).
- **`GrB_NOT_IMPLEMENTED`** – if G->A has a user-defined type.
- **`LAGRAPH_IO_ERROR`** – if the file could not be written to.

Returns
any GraphBLAS errors that may have been encountered.

int **`LAGraph_Matrix_Print`**(const GrB_Matrix A, *LAGraph_PrintLevel* pr, FILE *f, char *msg)

LAGraph_Matrix_Print displays a matrix in a human-readable form. This method is not meant for saving a GrB_Matrix to a file; see *LAGraph_MMWrite* for that method.

Parameters

- **`A`** – **[in]** matrix to display.
- **`pr`** – **[in]** print level.
- **`f`** – **[inout]** handle to an open file to write to.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if A or f are NULL.
- **`GrB_NOT_IMPLEMENTED`** – if A has a user-defined type.
- **`LAGRAPH_IO_ERROR`** – if the file could not be written to.

Returns
any GraphBLAS errors that may have been encountered.

int **`LAGraph_Vector_Print`**(const GrB_Vector v, *LAGraph_PrintLevel* pr, FILE *f, char *msg)

LAGraph_Vector_Print displays a vector in a human-readable form. This method is not meant for saving a GrB_Vector to a file. To perform that operation, copy the GrB_Vector into an n-by-1 GrB_Matrix and use *LAGraph_MMWrite* .

Parameters

- `v` – **[in]** vector to display.
- `pr` – **[in]** print level.
- `f` – **[inout]** handle to an open file to write to.
- `msg` – **[inout]** any error messages.

Return values

- `GrB_SUCCESS` – if successful.
- `GrB_NULL_POINTER` – if v or f are NULL.
- `GrB_NOT_IMPLEMENTED` – if v has a user-defined type.
- `LAGRAPH_IO_ERROR` – if the file could not be written to.

Returns

any GraphBLAS errors that may have been encountered.

enum **`LAGraph_PrintLevel`**

LAGraph_PrintLevel: an enum to control how much to print in LAGraph_*_Print methods.

Values:

enumerator **`LAGraph_SILENT`**

nothing is printed.

enumerator **`LAGraph_SUMMARY`**

print a terse summary.

enumerator **`LAGraph_SHORT`**

short description, about 30 entries.

enumerator **`LAGraph_COMPLETE`**

print the entire contents of the object.

enumerator **`LAGraph_SHORT_VERBOSE`**

short, but with "%.15g" for doubles.

enumerator **`LAGraph_COMPLETE_VERBOSE`**

complete, but "%.15g" for doubles.

5.6 Matrix/Vector Generation

int **`LAGraph_Random_Seed`**(GrB_Vector State, uint64_t seed, char *msg)

LAGraph_Random_Seed creates a pseudo-random vector containing an array of different pseudo-random streams, one per entry. On input, the values of the State vector are ignored but its structure is used. On output, all entries that were in the original structure of the State vector are assigned random values, depending on the scalar seed value. Each entry is considered its own pseudo-random number stream, with the overall seed value being revised for each entry in the vector, depending on their index in the vector.

More precisely, if the entry State [i] is present in the State vector, it is initialized with the pseudo random number State [i] = splitmix64 (i + seed); see https://dl.acm.org/doi/10.1145/2714064.2660195 for details, or https://en.wikipedia.org/wiki/Xorshift .

To call this method again with a new scalar seed, for subsequent iterations for the same State vector, it is advisable to advance the seed by at least n, where n is the dimension of the State vector. Otherwise, the random number streams will be correlated. For example, if seed++ is performed when this method is called again, then the new State [1] stream will be identical to the prior State [0] stream.

The State vector should normally be of type GrB_UINT64, but this is not enforced. Typecasting will be performed if it has a different type, which will affect the pseudo-random numbers generated and results are thus not guaranteed in this case.

Parameters

- **`State`** – **[inout]** vector to initialize with pseudo-random numbers.
- **`seed`** – **[in]** scalar seed value.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if State is NULL.

Returns

any GraphBLAS errors that may have been encountered.

int **`LAGraph_Random_Next`**(GrB_Vector State, char *msg)

LAGraph_Random_Next takes as input a vector previously initialized by LAGraph_Random_Seed, and modifies all its entries so that they take on their next value in their respective pseudo-random number streams.

Each stream in State [i] should be initialized by LAGraph_Random_Seed, and then advanced to the next pseudo-random value with LAGraph_Random_Next, which computes State [i] = xorshift64 (State [i]). See https://doi.org/10.18637/jss.v008.i14 and https://en.wikipedia.org/wiki/Xorshift .

Parameters

- **`State`** – **[inout]** vector with random numbers to be advanced.
- **`msg`** – **[inout]** any error messages.

Return values

- **`GrB_SUCCESS`** – if successful.
- **`GrB_NULL_POINTER`** – if State is NULL.

Returns

any GraphBLAS errors that may have been encountered.

5.7 Pre-defined semirings

LAGraph adds the following pre-defined semirings. They are created by *LAGr_Init* or *LAGraph_Init*, and freed by *LAGraph_Finalize*.

- **LAGraph_plus_first_T:**
  Uses the *GrB_PLUS_MONOID_T* monoid and the corresponding *GrB_FIRST_T* multiplicative operator:

```
LAGraph_plus_first_int8
LAGraph_plus_first_int16
LAGraph_plus_first_int32
LAGraph_plus_first_int64
LAGraph_plus_first_uint8
LAGraph_plus_first_uint16
LAGraph_plus_first_uint32
LAGraph_plus_first_uint64
LAGraph_plus_first_fp32
LAGraph_plus_first_fp64
```

- **LAGraph_plus_second_T**
  Uses the *GrB_PLUS_MONOID_T* monoid and the corresponding *GrB_SECOND_T* multiplicative operator:

```
LAGraph_plus_second_int8
LAGraph_plus_second_int16
LAGraph_plus_second_int32
LAGraph_plus_second_int64
LAGraph_plus_second_uint8
LAGraph_plus_second_uint16
LAGraph_plus_second_uint32
LAGraph_plus_second_uint64
LAGraph_plus_second_fp32
LAGraph_plus_second_fp64
```

- **LAGraph_plus_one_T:**
  Uses the *GrB_PLUS_MONOID_T* monoid and the corresponding *GrB_ONEB_T* multiplicative operator:

```
LAGraph_plus_one_int8
LAGraph_plus_one_int16
LAGraph_plus_one_int32
LAGraph_plus_one_int64
LAGraph_plus_one_uint8
LAGraph_plus_one_uint16
LAGraph_plus_one_uint32
LAGraph_plus_one_uint64
LAGraph_plus_one_fp32
LAGraph_plus_one_fp64
```

- **LAGraph_any_one_T:**
  Uses the *GrB_MIN_MONOID_T* for non-boolean types or *GrB_LOR_MONOID_BOOL* for boolean, and the *GrB_ONEB_T* multiplicative op.

  These semirings are very useful for unweighted graphs, or for algorithms that operate only on the sparsity structure of unweighted graphs:

```
LAGraph_any_one_bool
LAGraph_any_one_int8
LAGraph_any_one_int16
LAGraph_any_one_int32
LAGraph_any_one_int64
LAGraph_any_one_uint8
LAGraph_any_one_uint16
```

(continues on next page)

(continued from previous page)

```
LAGraph_any_one_uint32
LAGraph_any_one_uint64
LAGraph_any_one_fp32
LAGraph_any_one_fp64
```

CHAPTER

SIX

EXPERIMENTAL ALGORITHMS

LAGraph includes a set of experimental algorithms and utilities, in the LAGraph/experimental folder. The include file appears in LAGraph/include/LAGraphX.h. These methods are in various states of development, and their C APIs are not guaranteed to be stable. They are not guaranteed to have all of their performance issues resolved. However, they have been tested, debugged, and mostly benchmarked.

New algorithms and utilities can be contributed by placing them in the experimental/algorithm or experimental/utility folder. Tests for new algorithms or utilities should be placed in the experimental/test folder, and benchmark programs that exercise their performance (and typically check results) should be placed in the experimental/benchmark folder.

An simple example algorithm and its test and benchmark is provided, which serves as a template for creating new algorithms:

- **algorithm/LAGraph_HelloWorld.c**
  a simple "algorithm" that merely creates a copy of the G->A adjacency matrix.
- **benchmark/helloworld2_demo.c**
  a benchmark program illustrates how to write a main program that loads in a graph, calls an algorithm, and checks and prints the result. If any file appears in the benchmark folder with a name ending in _demo.c, then the CMake script will find it and compile it.
- **benchmark/helloworld_demo.c**
  another benchmark program. This one relies on internal utilities. See the file for details.
- **test/test_HelloWorld.c**
  a test program, using the acutest test suite. If any file appears in experimenta/test with the prefix test_*, the CMake script will compile it and include it in the "make test" target.

CHAPTER
SEVEN

INSTALLATION

LAGraph is available at https://github.com/GraphBLAS/LAGraph. Be sure to check out the default `stable` branch, or use one of the stable releases. LAGraph requires SuiteSparse:GraphBLAS, available at https://github.com/DrTimothyAldenDavis/GraphBLAS.

To compile and install LAGraph, you must first compile and install a recent version of SuiteSparse:GraphBLAS. Place LAGraph and GraphBLAS in the same folder, side-by-side. Compile and (optionally) install SuiteSparse:GraphBLAS (see the documentation in SuiteSparse:GraphBLAS for details). At least on Linux or Mac, if GraphBLAS is not installed system-wide, LAGraph can find it if GraphBLAS appears in the same folder as LAGraph, so you do not need system privileges to use GraphBLAS.

LAGraph includes a *CMakeLists.txt* file that does the bulk of the work to build the package. Also included is a very simple *Makefile* that simplifies the use of *make* for Linux and MacOS. In Linux or Mac, you can use it to run these commands:

```
cd LAGraph
make
make test
```

If you have system admin privileges, you can then install LAGraph:

```
sudo make install
```

On Windows, the *CMakeLists.txt* file can be imported into MS Visual Studio, and LAGraph can be built directly from there.

CHAPTER EIGHT

ACKNOWLEDGEMENTS

This work is funded in part by:

1. The United States Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. [DM22-0790]
2. The United States Department of Defense under Contract No. FA8650-18-2-7835 and HR0011-18-3-0007
3. NVIDIA, Intel, MIT Lincoln Laboratory, MathWorks, IBM, Julia Computing, Redis, and FalkorDB.
4. The National Science Foundation (1514406, OAC-1740333, CCF-1629657).
5. The United States Department of Energy, Office of Science, ASCR Contract No. DE-AC02-05CH11231.

CHAPTER

NINE

REFERENCES

- Graph algorithms in the language of linear algebra, Jeremy Kepner and John Gilbert, SIAM, 2011.
- Mathematical Foundations of the GraphBLAS, Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluc, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Jose Moreira, John D. Owens, Carl Yang, Marcin Zalewski, Timothy Mattson, IEEE High Performance Extreme Computing, 2016
- Design of the GraphBLAS API for C, Aydin Buluc, Tim Mattson, Scott McMillan, Jose Moreira, and Carl Yang Graph Algorithms Building Blocks workshop at IPDPS, 2017
- LAGraph: A Community Effort to Collect Graph Algorithms Built on Top of the GraphBLAS T Mattson, TA Davis, M Kumar, A Buluç, S McMillan, J Moreira, C Yang GrAPL workshop at IPDPS 2019. https://people.eecs.berkeley.edu/~aydin/LAGraph19.pdf
- LAGraph: Linear Algebra, Network Analysis Libraries, and the Study of Graph Algorithms, Gabor Szarnyas, David A. Bader, Timothy A. Davis, James Kitchen, Timothy G. Mattson, Scott McMillan, Erik Welch, GrAPL workshop at IPDPS 2021
- Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra Timothy A. Davis, ACM Trans. Math. Softw., vol 45, no 4, Dec. 2019. https://doi.org/10.1145/3322125
- Hoke, Tanner (2022). An Implementation of Fast Graphlet Transform in GraphBLAS. Undergraduate Research Scholars Program, Texas A&M University. https://hdl.handle.net/1969.1/196609
- Konduri, Pranav S (2022). An Implementation of the Parallel K-core Decomposition Algorithm in GraphBLAS. Undergraduate Research Scholars Program, Texas A&M University. https://hdl.handle.net/1969.1/196516
- Waheed, Abeer (2022). TriPoll in GraphBLAS. Undergraduate Research Scholars Program, Texas A&M University. https://hdl.handle.net/1969.1/196576

CHAPTER TEN

EXAMPLE USAGE

Note that this simple example does not check any error conditions.

```c
#include "LAGraph.h"

int main (void)
{
    // initialize LAGraph
    char msg [LAGRAPH_MSG_LEN] ;
    LAGraph_Init (msg) ;
    GrB_Matrix A = NULL ;
    GrB_Vector centrality = NULL ;
    LAGraph_Graph G = NULL ;

    // read a Matrix Market file from stdin and create a graph
    LAGraph_MMRead (&A, stdin, msg) ;
    LAGraph_New (&G, &A, LAGraph_ADJACENCY_UNDIRECTED, msg) ;

    // compute the out-degree of every node
    LAGraph_Cached_OutDegree (G, msg) ;

    // compute the pagerank
    int niters = 0 ;
    LAGr_PageRank (&centrality, &niters, G, 0.85, 1e-4, 100, msg) ;

    // print the result
    LAGraph_Vector_Print (centrality, LAGraph_COMPLETE, stdout, msg) ;

    // free the graph, the pagerank, and finish LAGraph
    LAGraph_Delete (&G, msg) ;
    GrB_free (&centrality) ;
    LAGraph_Finalize (msg) ;
}
```

genindex

INDEX