GraphBLAS C Test Infrastructure

Introduction

This document contains a high-level description of a C-based test infrastructure for the GraphBLAS library of sparse matrix routines. The test suite is designed to verify correctness of all operations and methods in the library for the built-in GraphBLAS objects as well as sample user-defined objects. Inputs and outputs are in Matrix Market format and data can be generated by MATLAB/Octave scripts provided. The test suite uses the SuiteSparse GraphBLAS library to generate the expected outputs. To generate those outputs, the test codes are linked with the SuiteSparse library and run with a flag to denote data generation mode. To test another library, the same test codes are linked with that library and run without the data generation flag. In generate mode, GraphBLAS functions are written on specific inputs and output data are written to the data directory in the test infrastructure. In test mode, GraphBLAS functions are run on the same inputs after which the result is compared to the answer file in the data directory. Data generation is done from the CTest directory using the Makefile supplied in that directory. Library testing is done in whatever build directory contains the test library and uses cmake in that build directory as per the top-level README instructions.

GraphBLAS Interface

GraphBLAS contains a set of objects with methods for using those objects as well as a set of operations on those objects. Each GraphBLAS operation has the following inputs:

- An input/output matrix for which initial values CAN be supplied.
- One, two, or three inputs depending on the operation to be tested.
- An optional mask that can be used to fill selectively the output data.
- An optional accumulator that can be used with the initial values.
- An optional descriptor that controls various options in running the operation.

In the test infrastructure, these inputs are captured in the `testargs` C structure that is input to each code in the test suite. The structure contains all possible arguments that might be used in calling a GraphBLAS function and each test code selects the arguments that are to be used in the test run for the operation being tested. If an argument is not present in the structure, then it is not used in running the test. If a required input for any test is not present in the structure, the test is run on a sample user-defined object.

To specify built-in objects to be used for a particular test, the test infrastructure contains a database of GraphBLAS objects. This database is created using the Python script codegen.py in the util directory. The database consists of arrays of GraphBLAS built-in objects that are indexed to access the desired object. The Python script generates the arrays containing the objects as well as the accessor functions needed to retrieve them. The script is written from scratch using the GraphBLAS include file GraphBLAS.h in the SuiteSparse distribution. The current version used in the Python script is 3.2; updating to 3.3 will requiring manually updating the arrays in the Python script that are used to generate the database code. The database code is linked into the test suite so that all test drivers can access the database.

The `testargs` structure includes an array `specobj` that specifies the index of a built-in object to be used as input for the GraphBLAS function. The `specobj` array is indexed by object type; the object type tells which database array to access using the index provided; the test codes demonstrate usage. There are 8 possible objects that can be accessed in the database, as shown in the following enumerator:

```
typedef enum { TYPE, SEMI, MON, BINOP, UNOP, SELOP, DESC, ACCUM, TOTAL
} spec;
```

The `testargs` structure is shown below:

```
typedef struct testargs {      // input to all test driver codes
  bool generate;               // flag to denote data generation mode
  int specobj[TOTAL];          // indices for GraphBLAS object database
  char mask[MATNAME_SIZE];     // mask input file name base
  char input0[MATNAME_SIZE];   // first input file name base
  char input1[MATNAME_SIZE];   // second input file name base
  char input2[MATNAME_SIZE];   // third input file name base
  char output[MATNAME_SIZE];   // output file name base
  char initvals[MATNAME_SIZE]; // initial values file name base
  char testbase[MATNAME_SIZE]; // directory name for output data
  char inbase[MATNAME_SIZE];   // directory name for input data
  char spectest[MATNAME_SIZE]; // spec file name base
} testargs;
```

The values in the `specobj` array are indices into the database of objects of the specified type. For example, `specobj[SEMI]` is an index into the array of semirings supported by the test infrastructure; these semirings have the names specified in the SuiteSparse documentation and implemented in GraphBLAS.h. In a similar manner, the other values in the `specobj` array are type, monoid, binary operator, unary operator, select operation, descriptor, and accumulator (which is also a binary operator). These values all index into the appropriate data structure for the type of object. If the value in the `specobj` array is -1 for any object, then NULL is used for that object. If an object is required for the test and not supplied in the `specobj` array, then the user-defined object will be used instead; this can be used to test user-defined objects for a given operation.

The character arrays are the base names of files for each matrix required by the operation. Inputs 0-2 are in left to right order. If the mask name is the empty string, then the NULL mask is used. If `initvals` string is supplied, the output matrix is initialized with those values (required for accumulators). The `testbase` is the directory name for the outputs; this will be a subdirectory of the data directory (default is the name of the test being run). The `inbase` is the directory name for the inputs; this is also a subdirectory of the data directory (default is testread). Finally, the `spectest` string specifies a file that can be used to specify a test iteration.

The actual file names used for each test are created from the file base name and the indices of the objects being used. For example, if output “CD” is specified and the test is run with descriptor 3, binary operation 5, accumulator 7, and type 10 the output file name will be “CD_B5_D3_A7.mm” (N.B., accumulators are binary operations and are indices into the same array in the database). Input arrays are all of type double; the test infrastructure reads them and does a type conversion to produce the input type required for the test. The letters used as object designators for output files are as follows, where the designator is only in the file name if it is used: B=binary operator, S=semiring, M=monoid, U=unary operator, D=descriptor, L=select operation, A=accumulator (binary operator), T=type.

Test Iterations

The test suite provides the capability to loop over multiple objects in the database, running the test program on each of these objects to produce multiple tests in one run of the program. The loop is specified in a specification file that has the ".spec" extension. The specification file consists of a line for each input to the test driver in the format:

<SPEC> <number of objects in test> <list of object indices>

If any line consists of <SPEC> <N>, where N is greater than the number of objects in the database of that type, then all objects are used. If N is less than that number, then there must be a list of indices of size N on the same line. The possible values for <SPEC> and the maximum values are:

- TYPE 11	// all GraphBLAS built-in types
- SEMI 1355	// all GraphBLAS built-in semirings
- MON 55	// all GraphBLAS built-in monoids
- BINOP 300	// all GraphBLAS built-in binary operators
- UNOP 67	// all GraphBLAS built-in unary operators
- SELOP 16	// all GraphBLAS built-in select operators
- DESC 32	// all GraphBLAS built-in descriptors
- ACCUM <no max>	// accumulators to be used in iteration

The specification file also contains the base file name strings for all input and output files as follows:

- MASK <str>	// basename of the mask data file
- INPUT0 <str>	// basename of the left most input data file
- INPUT1 <str>	// basename of the next left most input data file
- INPUT2 <str>	// basename of the final input data file
- INIT<str>	// basename for initial values data file
- OUTPUT <str>	// basename for output; object-specific strings will be appended

The test iteration is a double or triple nested loop depending on test type. The innermost loop is over descriptors: each test case is run with all descriptors specified in the spec file or with NULL is none are specified. The next outer loop is over the primary test iteration object, which is either TYPE, SEMI, MON, BINOP, UNOP. The test loop selects the first one provided in that order. At this level, the primary test iteration object is paired with the accumulators listed in the spec file for the test execution. If an accumulator is -1 at any position in the list, NULL is used in that iteration. If the list of accumulators is shorter than the primary object list, then NULL is used for the latter part of the primary iteration. Finally, the outermost loop is only present for select operators, for which the primary iteration is over type (since select operators can operate on all types).

If a test does not have a value for the object needed in the test, then a user-defined object is used. The user-defined object is hard-coded into the test and uses the GrB_FP32 data type. There are user-defined semiring, monoid, binary operator, unary operator, type, and select operator objects. This can be used to test user-defined objects for all GraphBLAS operations.

A sample spec file is shown below, for the textmxm test (matrix-matrix multiply). This spec file specifies an iteration over 11 semirings (ANY_BOOL and 10 PLUS_TIMES), 4 descriptors, with 11 accumulators (PLUS for all types), input 0 and 1 (required for testmxm), initial values for the accumulation, and output file base.

```
SEMI 11 1314 352 357 362 367 372 377 382 387 392 397
DESC 4 0 1 2 3
ACCUM 11 50 51 52 53 54 55 56 57 58 59
INPUT0 A
INPUT1 B
OUTPUT CD
INIT A
```

Test Execution

Each test takes 2 arguments and options that control how the test is run. The options for each test are:

-h	print help
-g	run test in generate mode
-t <N>	index of a type object in the database
-u <N>	index of a unary operator in the database
-b <N>	index of a binary operator in the database
-s <N>	index of a semiring in the database
-m <N>	index of a monoid in the database
-l <N>	index of a select operator in the database
-d <N>	index of a descriptor in the database
-a <N>	index of a binary operator to be used as an accumulator
-m <str>	base file name for mask
-0 <str>	base file name for input 0 (left most)
-1 <str>	base file name for input 1 (next in left to right order)
-2 <str>	base file name for input 2 (next in left to right order)
-o <str>	base file name for output
-I <str>	base file name for initial values
-p <str>	file path name for spec file to use for a test iteration

A single test can be run by supplying the required indices. Index values can be found by running the codegen.py script with two arguments: <SPEC> and <ITEM>, where <ITEM> can be either a string (name of GraphBLAS built-in object of type <SPEC>) in which case the index is printed; or an index value in which case the name of the GraphBLAS built-in object is printed. If codegen.py is run with no arguments, the database code is generated in the util directory, which consists of several C files that implement the arrays and accessor functions that are linked into each test.

Alternatively, a spec file as specified above can be provided using the -p option, which directs the test to perform a test iteration as specified in the file. If both options and spec file are specified, then a subset of the test iteration is performed according to the values in the options. If no spec and no output file are supplied, then the test codes read a special “*.list” file with list extension. The list file contains any number of lines, each of which contains a full path name of a spec file. Each spec file is run through the test code; in this way, multiple test iterations can be performed at once.

In the Makefile, any test can be run using the “make N=<str> run” command, where testN is the test that will be run. To generate truth data, use the command “make N=<str> gen”. Alternatively, “make runall” and “make genall” run and generate for all tests, respectively.

Creating Tests

Spec and list files are created with a Python script specgen.py in the util directory. That script uses the same database constructor arrays to create spec files in the format described above. The script can generate spec and list files for three different suites: working, regression, and full. The working suite is what is currently being debugged. The regression suite is what currently passes for all builds. The full suite is kept only for refence since it is so large and specifies iterations over all objects in GraphBLAS. These test suites do **not** need to be generated, since they are checked into the git repository. However, some familiarity with the test suite generation process is desirable so that new tests can be added.

The test suite is generated via a command-line argument to the script. A single spec file can also be generated via command-line arguments if desired. In the CTest directory, run:

- "python util/specgen.py" to print a help message
- "python util/specgen.py WORKING" to generate spec and list files for the current working tests; these will be put in the data/working directory
- "python util/specgen.py REGRESSION" to generate spec and list files for the regression test suite; these will be put in the data/regression directory
- "python util/specgen.py FULL" to create test specs that iterate over all objects; these will be put in the data/fullspec directory
- "python util/specgen.py f out in0 in1 in2 init mask accum-list obj-name obj-list desc" to generate a single spec file and write to current directory; the desc parameter can be a number specifying all descriptors up to that index. The value "ALL" can be used to denote all objects of that type.
    - For example: "python util/specgen.py data/specfiles/testmxm CB A B '' A M 'ANY_BOOL,PLUS' 'SEMI', 'ANY_PAIR_BOOL,PLUS_TIMES' ALL" to generate one spec file for mxm with output prefix CB, inputs A, B, initial values A, mask M, accumulator list that includes ANY_BOOL and all of the PLUS operators, a semiring list that includes ANY_PAIR_BOOL and all of the PLUS_TIMES semirings, and all descriptors

To switch between the test suites, change the "specfiles" link in the data directory to point to the desired test suite before running the codes. There is a special Makefile target in the CTest directory that accomplishes this. To change to the working test suite, use "make workingspec" and to change to the regression test suite, use "make regressionspec". To add a new test suite, the function must be added to the Python script (the existing functions can be used as a guide) and the call to that function must be added to the "main" routine similarly to the others.

The code for implementing the test loops, parsing test arguments, reading,writing, and checking matrices, etc. is all contained in the util directory, along with the python scripts for generating and querying the database. The README in the CTest directory outlines the process for setting up the test infrastructure and generating the input and reference data. The test infrastructure assumes that a working SuiteSparse/GraphBLAS implementation exists at the path specified in the Makefile.

Directory Structure

The test infrastructure is contained in the directory CTest. In that directory there are:

- C files test*.c for each GraphBLAS operation (named cryptically but appropriately)
- Makefile: for generating data and running tests. To run a test, do "make N=<STR> run", where test<STR>.c is a file in the directory. To generate data for that test, do "make N=<STR> gen". To run or generate all tests, do "make runall" or "make genall" respectively. The latter will produce

files *.out in all the data directories to log exactly what objects were used. No arguments are used with these make targets, so the default test is run for all tests.

- utils/: the utils directory contains support routines and scripts for the test infrastructure, including "test_utils.h" which must be included to use the tests. The Makefile has all the appropriate flags for building the tests. To generate the library, do "make lib" at the CTest level.
- data/: the data directory contains inputs and outputs for the tests. The subdirectory testread is the default location for inputs; all inputs are re-used for all tests. The exceptions are the assign tests; the defaults for these tests use outputs from the extract tests (therefore the genall target consists of two passes: the extract data is generated before all the rest). There are links within the testread directory for the extract test outputs. The other subdirectories contain the outputs for each test and are named for the executable, which is the default parameter for the output file basename.
- data/specfiles: contains all list and spec files for running tests; these are used if no arguments to tests are supplied.
- data/fullspec contains all list and spec files for a full iteration over all GraphBLAS objects
- data/regression contains all list and spec files for regression tests (i.e., all passing)

Data files

Data files can be created by MATLAB, Octave, or C or any program that writes Matrix Market format. There are constraints on the types of files that the test infrastructure uses, those are built into the file read process. MATLAB/Octave scripts for generating data according to those constraints are contained in the CTest directory.

The test infrastructure contains three primary ways to read and write data. The first is "{read, write}_matlab_{vector, matrix}". With these routines, a Matrix Market file from MATLAB is read; the assumption with this file is that it contains a sparse matrix written directly from MATLAB and is therefore double precision or Boolean (the only types supported by sparse matrices). The read routine will read the file into the GraphBLAS type specified by the argument, typecasting as necessary during the read.

The second type of file access is "{read, write_typed_{vector, matrix}". With these routines, a Matrix Market file that has been written by the test infrastructure is read; the assumption with this file is that it contains a sparse matrix that is neither symmetric or pattern and that it contains data of the type specified in the input. The file name is generated from the base filename input and will have a string appended to denote the data type. For example, if called with base name "C" and type GrB_BOOL, the filename will be "C_bool.mm".

The final type of file operation is "read_test_index". This routine is used on a Matrix Market file that contains only dense vector data (either row or column). This file will contain test indices for extract and assign operations and must have values within the required range for data access. Four special cases are possible: 1) if "ALL" is used as the file name, then no file is read and the GrB_ALL index is used; 2) if the file name contains the string "RANGE" then the GrB_RANGE index is used and the file must contain 2 values in BEGIN, END order; 3) if the file name contains the string "STRIDE" then the GrB_STRIDE index is used and the file must contain 3 values in BEGIN, END, INC order; and 4) if the file name contains the string "BACK" then the GrB_BACKWARDS index is used and the file must contain 3 values in BEGIN, END, INC order.

For each of the first two types of file access, there are also “check_...” routines for checking the file against a GraphBLAS matrix. In generate mode, the output files are written using the “write_...” routines; in test mode, the “check_...” routines are used to compare matrices against data files that have already been written. Since the indices are constant in the database and file names are consistently generated, the check routines will always find the right file for the comparison.