Chapter 1

STINGER: Spatio-Temporal Interaction Networks and Graphs Extensible Representation

Contributed by Tim Shaffer

1.1 Background

STINGER [1] is a general-purpose graph data structure that supports streaming updates and is designed for efficient parallel processing. Social network interactions, business intelligence, and other live data sources can produce an an enormous number of events. To process these and other more static applications such as those in bioinformatics, it is necessary to handle extremely rapid updates and large graphs. In addition, using the collected graph data effectively requires streaming and parallel algorithms suitable for use online at these scales. STINGER is designed to handle millions of updates per second on commodity hardware, and graph sizes of millions to billions of edges and vertices.

In addition to the graph primitives themselves, the implementation of STINGER comes with a number of tools and algorithms for working with graphs. These include streaming clustering coefficients & connected components, parallel agglomerative clustering, and a number of other algorithms. STINGER also allows direct access to the graph primitives in parallel for implementing custom algorithms efficiently.

STINGER was primarily developed at the Georgia Institute of Technology. STINGER is implemented as a C library that can be linked into other programs, and comes with bindings for Python. STINGER is also bundled with an RPC server to allow using it without linking the library directly into applications, or over the network from multiple clients. The project web site is at `http://www.stingergraph.com/`. The source code is available on Github at `https://github.com/stingergraph/stinger`.

1.2 Expressing Graphs

STINGER operates on sparse, in-memory representations of directed graphs. Applications may write checkpoints to disk to resume processing after a failure or other interruption. The API is designed to support dynamic changes to the graph, allowing for additions, changes to weights, and

removals of vertices and edges. STINGER does not impose significant restrictions on the structure or properties of the graph. Thus users have flexibility in expressing domain-specific information as graphs.

Because of its focus on performance and scalability, STINGER does not allow elements of the graph to contain arbitrary amounts of data. STINGER is, however, designed to support arbitrary numbers of edges and high-degree vertices. Vertices are tagged with type and weight. Edges are tagged with type, weight, and timestamps. These fields are required to fit in 64 bit integers, ensuring compact data structures. Using vertex and edge types it is possible to encode multiple "layers" of information in the graph. Thus rather than allowing extensive annotations on graph elements, STINGER encourages users to encode additional information as different vertex and edge types in the same graph.

1.3 Syntax

The reference implementation of STINGER is written as a C library. Documentation generated from the header files is available at `http://www.stingergraph.com/doxygen/index.html`. There are functions to perform basic operations on the graph, e.g. inserting edges,

```
int stinger_insert_edge ( struct stinger *  G,
        int64_t  type,
        int64_t  from,
        int64_t  to,
        int64_t  weight,
        int64_t  timestamp
    )
```

setting vertex weights,

```
int64_t stinger_set_vweight ( struct stinger *  S_,
        int64_t  i_,
        int64_t  weight_
    )
```

removing all edges of a particular type,

```
void stinger_remove_all_edges_of_type ( struct stinger *  G,
        int64_t  type
    )
```

and so on. Naming of functions and arguments is fairly self-explanatory. STINGER's API also includes a number of convenience functions, such as `stinger_incr_edge_pair` and related functions for working with edge pairs to simulate an undirected graph. The API includes additional functions for checking the in-degree/out-degree of a vertex, counting the total number of edges in a graph, and other operations on the low-level graph primitives and the graph itself.

STINGER provides a set of macros for performing some operation on all vertices, edges, etc.

```
STINGER_FORALL_EDGES_OF_VTX_BEGIN ( STINGER_,
        VTX_
    )
```

For example, the above macro marks a block of code to be run for all edges of a given vertex. This macro block must be closed by a corresponding `STINGER_FORALL_EDGES_OF_VTX_END` macro. There are corresponding macros for iterating over all edges of a particular type, all vertices, and so on. It is possible to specify a filter, such as iterating only over edges modified after some specified time.

Utility functions are provided for batch inserts, reading/writing data on disk, translating to/from other formats such as CSR, and related operations. A number of algorithm implementations are included with STINGER as well. These make use of the core operations discussed above. Each algorithm implementation has its own interface and requirements, so it is difficult to give a summary here.

1.4 Key Graph Primitives

STINGER's core graph primitives are vertices and edges. These primitives can be annotated with limited information such as weights and types. All graphs in STINGER are directed, but convenience functions are provided for simulating undirected graphs. STINGER's low-level primitives are intended to be used directly in implementing graph algorithms. Thus there is no need for STINGER to include a query language or other abstractions. Instead, STINGER focuses on a performant and user-friendly interface to its few core primitives.

1.5 Execution Model

A key design consideration for STINGER is efficient parallel execution. STINGER assumes a multicore, shared memory machine. This could be a commodity desktop machine, a large server, or a supercomputer such as Cray XMT2. STINGER's iteration macros automatically take advantage of OpemMP to parallelize tasks. In addition, the STINGER data structure itself has several design features to enable parallel execution. The vertices of the graph are stored in an array, so that it is trivial to parallelize iteration over all vertices. All adjacency lists are stored in fixed-sized blocks where each block contains only edges of a single type. For insertions and removals, it is possible to split and merge edge blocks without reallocating the entire edge list. When iterating over all edges incident to a vertex, each block is processed in parallel. STINGER also maintains global edge lists to allow iteration over all edges in the graph. This allows iteration over edges without first searching through vertices. Each edge type used in the graph is stored in its own global list. The global edge list for a particular type points to every block containing edges of that type. Thus it is possible to iterate only over edges of a given type, or to iterate over all edge types in parallel. Just as when iterating over edges incident to a vertex, the blocks of edges allow for easy parallelization. STINGER does not provide full ACID semantics. It does, however, ensure consistency when using the parallel iteration macros.

1.6 Examples

This example, taken from the official release at the time of writing, demonstrates computing the approximate and exact diameter of a graph. It is written in C++ and uses an implementation of Dijkstra's algorithm (also provided) to find shortest paths in the graph.

```
//
// Created by jdeeb3 on 5/24/16.
//
```

```
#include "diameter.h"
#include "shortest_paths.h"
/**
 * this algorithm gives an approximation of the graph diameter.
 * It works by starting from a source vertex, and finds an end vertex that is farthest away
 * This process is repeated by treating that end vertex as the new starting vertex and ends
 * when the graph distance no longer increases
 * Inputs: S- the graph itself, nv - the total number of active verticies in the graph,
 * source - a starting vertex
 * dist - the variable that will hold the resulting diameter, and ignore_weights - a flag
 * that controls wheither a user
 * wants to consider the weights along the edge or not.
 */
int64_t pseudo_diameter(stinger_t * S,
        int64_t NV ,
        int64_t source,
        int64_t dist,
        bool ignore_weights){
    //int64_t source = 1; // start at the first vertex in the graph,
    //this could be any vertex
    dist = 0;
    int64_t target = source;

    std::vector<int64_t> paths(NV);
    while(1){
        int64_t new_source = target;
        paths = dijkstra(S, NV, new_source, ignore_weights);
        int64_t max = std::numeric_limits<int64_t>::min();
        int64_t max_index = 0;
        for( int64_t i = 0; i< NV; i++){
            if (paths[i] > max and paths[i] != std::numeric_limits<int64_t>::max()){
                max = paths[i];
                max_index = i;
            }
        }
        if (max > dist){
            target = max_index;
            //source = new_source;
            dist = max;
        }
        else{
            break;
        }
    }
    return dist;
}
/**
 * this algorithm gives an exact diameter for the graph.
```

```
 * It runs Dijkstras for every vertex in the graph, and returns the maximum shortest path
 * Inputs: S- the graph itself, nv - the total number of active verticies in the graph
*/

int64_t
exact_diameter(stinger_t * S, int64_t NV){
    std::vector<std::vector <int64_t> > all_pairs (NV, std::vector<int64_t>(NV));
    all_pairs = all_pairs_dijkstra(S, NV);
    int64_t max = std::numeric_limits<int64_t>::min();
    for( int64_t i = 0; i< NV; i++){
        for (int64_t j = 0; j < NV; j++){
            if (all_pairs[i][j] > max and all_pairs[i][j]
                    != std::numeric_limits<int64_t>::max()){
                max = all_pairs[i][j];
            }
        }
    }
    return max;
}
```

1.7 Conclusion

STINGER provides a powerful platform for working with large, dynamic graphs. The core primitives used in STINGER are general enough to support a wide variety of graph-based problems. STINGER has been used successfully in astrophysics, bioinformatics, and numerous other fields. The reference interface for STINGER is a C library, but Python bindings and an RPC server are provided as well. STINGER is designed to handle streaming updates and includes streaming and parallel implementation of a number of graph algorithms. The STINGER data structure is designed to allow fast updates as data changes over time, and to take advantage of parallel computing power when available. STINGER works on both commodity hardware and highly parallel supercomputers without extensive setup.

Bibliography

[1] David A Bader, Jonathan Berry, Adam Amos-Binks, Daniel Chavarría-Miranda, Charles Hastings, Kamesh Madduri, and Steven C Poulos. Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. *Georgia Institute of Technology, Tech. Rep*, 2009.