The code

The point

This code is meant to be used for simulation of a node process paired with an edge process, which will be referred to as a "temporal process" for lack of better terminology. A node process is a model for information transfer between nodes in time. An edge process is a model for edge removal and addition in time. This code is meant to aid in the study of how an edge process affects a node process.

This code allows from someone to run an experiment on a temporal process.

What do the classes mean

The **Experiment** class defines the parameters of the experiment: the number of *trials*, the *time each trial will take* (in terms of iterations), and *the measurement that will be conducted per iteration* (one could say per observation).

The **Experiment** class takes as input a **TemporalProcess**, which is the object one will be experimenting on. A **TemporalProcess** consists of an **eMTRG** and an **Updater**. The **eMTRG** class generates a sample path for a edge-Markovian temporal random graph. The **Updater** class updates the data stored at nodes based upon an inputted graph structure. One can think of the **Updater** as defining a limited kind of dynamical process on each fixed graph.

Finally, the **Network** class is simply a wrapper class for an edgelist with some useful functions.

How to use

Below is an example main method with comments:

```
int main() {

   // parameters for random graph model
   size_t nodes = 1000;
   double alpha = .001;
   double p = .001;

   // parameters for experiment
   size_t trials = 10;
   size_t num_identical = 10;

   // construct experiment object
   TotalerResidualExperiment<erTRG,ReachabilityUpdater>
experiment(TemporalProcess<erTRG,ReachabilityUpdater>(erTRG(nodes,alpha,p),ReachabilityUpdater(nodes)),t
rials,num_identical);

   // Run experiment
```

```
    experiment();
}
```

This code will create a whitespace separated output file “data.txt” with:

- a header giving info about the Updater, the eMTRG, and Experiment
- Rows for the different trials
- Columns for the different iterations
- Entries with the result of the measurement taken on each trial

By replacing the various class names in the above code with one’s own derived classes, one can run many different experiments.

Creating derived classes

Generally, one will wish to create derived classes of the Updated, the erRTG, and the Experiment classes. Below, it is indicated one one would do this.

Overview of Classes

There are five base classes with a few derived classes.

1. **Experiment<S,T>:** An object of this class is a particular “experiment” on a temporal process. The call operator for an Experiment object runs some number of *trials*, where each trial consists of some number of *iterations* of the temporal process. On each iteration, a *measurement* is done to summarize some information about that iteration of the temporal process. The measurement data is outputted to a file.
    a. **Member data:**
        i. **TemporalProcess<S,T> tp:** Process on which the experiment is run
        ii. **size_t trials:** number of trials to run
        iii. **ofstream fout:** filename for data output
    b. **Functions to specify in derived classes:**
        i. **double measurement(vector<unordered_map<size_t,size_t>>&& node_data):** this function gets some summary information about the temporal process at a particular iteration (MAKE IT SO MORE THAN 1 MEASURMENT CAN BE MADE PER ITERATION)
            1. **Parameters:**
                a. **Node_data:** the data structure containing the data owned by each of the nodes
            2. **Return:** The result of the measurement
        ii. **string info():** returns information about the experiment; i.e. what was the experimental setup?
        iii. **string specific_info():** returns information about the specific experiment; i.e. what kind of measurement was taken?
        iv. **void trial():** Executes a trial.
    c. **Supplied Derived classes:**

i. **IterationExperiment<S,T>:** Does an experiment where each trial is a fixed number of iterations.
    1. **Member data:**
        a. **size_t iter:** the number of iterations to run
    2. **Functions to specify in derived classes:**
        a. **string specific_info():** returns a string of info about the specific experiment; i.e. what kind of measurement was taken?
        b. **double measurement(vector<unordered_map<size_t,size_t>>&& node_data)**
    3. **Supplied Derived classes:**
        a. **TotalerIterationExperiment<S,T>:** Measurement function returns the total number of pieces of data held by the nodes
        b. **ValueTotalerIterationExperiment<S,T>:** Measurement function adds up the *values* held by the nodes

ii. **ResidualExperiment<S,T>:**
    1. **Member data:**
        a. **Size_t times_identical:** the number of consecutive iterations that have to result in the same entry for the trial to be considered over.
    2. **Functions to specify in derived classes:**
        a. **string specific_info():** see Experiment
        b. **double measurement(vector<unordered_map<size_t,size_t>>&& node_data)**
    3. **Supplied Derived classes:**
        a. **TotalerResidualExperiment<S,T>:** Measurement function returns the total number of pieces of data held by the nodes
        b. **ValueTotalerIResidualExperiment<S,T>:** Measurement function adds up the *values* held by the nodes

2. **Network:** An object of this class is a graph (so RENAME TO GRAPH)
    a. **Member data:**
        i. **Edgelist el:** The graph adjacency structure is stored as an std::set<std::pair<size_t,size_t>> to enable O(logn) insert, O(logn) deletion, and O(logn) search for a nodes in-edges (an in-edge of a node i is an edge (I,j)).
        ii. **Size_t sz:** The number of nodes in the graph.
    b. **Selected member functions:**
        i. pair<Edgelist::iterator,size_t> range(size_t i) const: returns the in-edges of a vertex i. The in-edges are specified by an iterator pointing to the first

in-edge, and an integer giving the number of in-edges. Because in-edges are stored in order, sequential in-edges can be obtained via ++.

3. **eMTRG:** An object of this class generates a sample path for an edge-Markovian Temporal Random Graph.
    a. **Member Data:**
        i. **Size_t sz:** The size of the graph
        ii. **Network net:** the *next* graph in the sample path. After construction, net is the initial graph, and after each call to the call operator, net is equal to the *next* graph in the sample path. (THIS MEANS 1 EXTRA GRAPH IS GENERATED!)
    b. **Functions to specify in derived classes:**
        i. **double P(const Edge& e):** returns the probability Pr(e present at timestep t | e not present at timestep t-1)
        ii. **double Q(const Edge& e):** returns the probability Pr(e not present at timestep t | e present at timestep t-1)
        iii. **void initializer():** sets net to be the initial graph
        iv. **string info():** returns string of information about the object
    c. **Supplied Derived classes:**
        i. **erTRG:** objects of this class generate Erdos-Renyi Temporal Random Graphs. For Erdos-Renyi Temporal Random Graphs, P(e) = alpha*p and Q(e) = alpha*(1-p), and the initial distribution is G(n,p).
            1. **Member data:**
                a. **double alpha**
                b. **double p**
    d. **Possible Further derived classes:**
        i. **clTRG:** objects of this class would generate a Chung-Lu Temporal Random Graph. Here we would need the expected in- and out- degree distributions, which would be stored as member variables of the object.
4. **Updater:** An object of this class defines the node process; i.e. how information is spread throughout a fixed graph.
    a. **Member data:**
        i. **Network net:** the network on which the update occurs. (MAKE SURE NOT CREATING COPIES ANYWHERE…)
        ii. **vector<unordered_map<size_t,size_t>> node_data:** data structure to store the data held by the nodes on the network. The i-th index of the vector holds the data for node i. The unordered map should be used such that node_data[i][j] is the information node i has that's associated with node j. An unordered_map instead of a vector is used because a node may not have data about all the other nodes.
    b. **Functions to specify in derived classes:**
        i. **vector<unordered_map<size_t,size_t>> operator()(Network&& net):** This function updates node_data based upon the graph net. The simplest example could be looping through all nodes and combining the data at

the current node with the data of it's neighbors. This combine procedure should happen in the combine function

ii. **void combine(size_t to, size_t from):** This function uses the information in *from* to update the information in *to*. Notice this function works one way: use from to update to; from is unmodified.

iii. **void node_initializer(size_t sz):** This function determines what data is held by each node initially (at t=0). This function should modify node_data.

iv. **void node_data_update(size_t i):** This function determines how data at a node should be updated as each step, irrespective of other nodes. For example, we could delete data that is too "old."

v. **void reset():** This function should reset member data in the Updater, which most likely will just involve running node_initializer.

vi. **string info():** returns a string of information about the updater

c. **Supplied Derived classes:**

i. **ComponentUpdater:** Updates nodes using only in-connectivity structure; i.e. node i is updated using data from node j only if and only if there is a directed path from j to i.

1. **Functions to be specified in derived classes:**

a. **void combine(size_t to, size_t from):** This function uses the information in *from* to update the information in *to*. Notice this function works one way: use from to update to; from is unmodified.

b. **void node_initializer(size_t sz):** This function determines what data is held by each node initially (at t=0). This function should modify node_data.

c. **void node_data_update(size_t i):** This function determines how data at a node should be updated as each step, irrespective of other nodes. For example, we could delete data that is too "old."

d. **string info():** returns a string of information about the updater

2. **Supplied Derived classes:**

a. **ReachabilityUpdater:** This class assigns a single piece of data to each node initially; i.e. the unordered_map contains a single entry i=>1 for node I (node_initializer). Node i will obtain all of node j's data, which is of the form (k_1=>1,k_2=>1,…,k_m=>1), if there is a directed path from j to i. Node_data_update is unused.

b. **ReachabilityPathUpdater:** This class assigns a single piece of data to each node initially; i.e. the unordered_map contains a single entry i=>1 for node I (node_initializer). If there is a directed path from node j to node i, then the values of any entries in node_data[j] and node_data[i]

with the *same* key are added together and given to node_data[i], and if node_data[j] has an entry whose key is not in node_data[i], then node_data[i] just receives that entry. Node_data_update is unused.

3. **Possible further derived classes:**
    a. **Deleting old paths:** Node_initializer assigns a i=>0 to node_data[i]. Node_data_update increments each element of node_data[i] by 1, deleting any data whose value is greater than some number. Combine compares entries with the same keys in node_data[i] and node_data[j], and assigns the MIN to node_data[i].

d. **Possible further derived classes:**
    i. **Taking data from neighboring nodes:** A class analogous to ReachabilityUpdater can be built except data is only sent from j to i if there is an edge from j to i.
        1. **Deleting old paths:** old data can be deleted as discussed above.

5. **TemporalProcess:** An object of this class consists of an eMTRG and an Updater. The call operator simply runs the Updater on the output of the eMTRG.
    a. **Template Parameters:**
        i. **S:** the eMRTG derived class used.
        ii. **T:** the Updater derived class used.
    b. **Member data:**
        i. **S emtrg**: the eMTRG derived class object.
        ii. **T updater**: the Updated derived class object.
    c. **Functions:**
        i. **vector<unordered_map<size_t,size_t>> operator()():** returns the node_data after update has been completed.
        ii. **Void reset():** resets the emtrg and updater objects.