# Pandemaniac Documentation

Angela Gong (anjoola@anjoola.com), Max Hirschhorn (visemet@outlook.com)
Updated June 2017 by Nikita Sirohi (nikita@thesirohis.com)

Source repositories:
- https://github.com/visemet/pandemaniac-graphui
- https://github.com/visemet/pandemaniac-modelsim

Note: All code is on the AWS instance. We haven't really been pushing these past few years…probably should. We tried to migrate all code to a shared cs144-miniprojects github, but had problems because of these existing repositories. So I guess figure out what makes the most sense. Try to keep this and the README's updated; they weren't really when we inherited this, and a lot of problems would have been resolved much faster had they been. Also, I tried to add to this document where possible/applicable rather than removing, but apologies if that makes something confusing. If you think it makes more sense to just remove old stuff/add what I wrote in its place, go for it. -- Nikita

## Contents

## 1 Web application

### Overview - Technology stack

*Redis*
Redis is an in-memory, key-value data store. The usage of this system is twofold.
Firstly, (and a very typical scenario), we use Redis to keep track of user sessions so that we can properly authenticate users when they make a request, e.g. to upload a submission. Another great feature of Redis is that the keys can be set to expire after a certain period of time. This allows us to enforce timed submissions, the actual mechanics of which are described in Appendix A.

*MongoDB*
MongoDB is a document-oriented database that stores data in a format similar to JSON (called

BSON, a *binary*-representation). The equivalent of a table (for those of you who are familiar with the relational model, e.g. from CS 121) is called a collection and is a potentially nested hierarchy of these documents. There is not a good reason *per se* of why to use this database over another for this type of application, other than out of my (Max) own convenience.

There are four (4) collections that the database will store.

teams: keeps track of the login credentials for each team.

| **Field** | **Type** | **Description** |
|---|---|---|
| `_id` | `ObjectId` | (default) |
| `name` | `string` | The team's name |
| `hash` | `string` | The hash of the team's password. Note that since we use bcrypt, the salt is automatically concatenated with the hash. |

graphs: keeps track of which graph file corresponds to what graph name, and its time of availability.

| **Field** | **Type** | **Description** |
|---|---|---|
| `_id` | `ObjectId` | (default) |
| `name` | `string` | The graph's name. Uses the semantic naming convention as described in the assignment to specify the number of players and the number of seeds nodes, e.g. "`1.4.4`". |
| `category` | `string` | The graph's category. Identifies the set of graphs for a particular day. Note that the webview always displays these in lexicographical order. |
| `file` | `string` | The file of the graph. This is the `.json` file that is served to the user when the send a download request. |
| `timeout` | `int` | The submission time limit for this graph. This is how long (in seconds) the user has in order to upload a valid submission (for it to be accepted). |
| `start` | `Date` | The time at which the graph becomes available for download, and hence submission. |
| `end` | `Date` | The time until which submissions for the graph are accepted. Note that under the current system, the students are able to download older graphs for their own practice use. |

attempts: keeps track of when teams have downloaded a graph, and when they have made valid submissions for it.

| Field | Type | Description |
|---|---|---|
| `_id` | `ObjectId` | (default) |
| `at` | `Date list option` | List of dates for when the user has successfully submitted. Omitted if the user has made no successful attempts at uploading a set of seed nodes. |
| `graph` | `string` | The graph for which the attempts were made. |
| `team` | `string` | The team that made the attempts. |

runs: keeps track of the information to be able to display the simulation. Note that the contents of the runs collection is only *read* by the web application, and written to by the simulation (more in the section "Simulation").

| Field | Type | Description |
|---|---|---|
| `_id` | `ObjectId` (default) | |
| `graph` | `string` | The graph used in the simulation. |
| `scores` | `map (string -> int)` | A map of team name to score. Map of the scores for each team that participate in the simulation. |
| `file` | `string` | The file that stores the diff for the nodes in the graph at each iteration. |
| `teams` | `string list` | The teams that participated in the simulation. |

*NodeJS*
Creates an Express server that exposes routes for the users (i.e. the students) to interact with.

Note/Update: I personally found the file structure corresponding to these pages confusing so… I wrote a bit about file structure in the README file for pandemaniac-graphui, where everything relevant to this section is. This is a pretty good description of what each of the website pages does. -- Nikita

- `/login`, `/logout`, and `/register` should all be self-explanatory for managing user authentication.
- / (the root page) displays the instructions for how to use the web application. To modify these instructions in any way, change the `views/index.jade` file. To clarify, jade is a templating language that gets transpiled to html. The instructions themselves are written in a markdown filter (*think* "block" or "section") of the template.
- `/submit` displays the list of graphs that are available for downloading. Each is a link of the form `/submit/:id` that shows the graph name.
- `/submit/:id` is where the students can download the graph or upload a submission for

it while time remains. After the student clicks the download button (refreshing the page once the download completes), they will see a shrinking bar to represent how much time remains and a file upload form to submit their seed nodes.

- `/submit/:id/download` and `/submit/:id/upload` are the actual routes responsible for serving the graph file to the student or handling the uploaded submission, respectively.
- `/graph` displays a series of tables (one for each *category* of graph types) that shows each teams score on the various graphs (if submitted) with a link of the form `/graph/:id` to view the simulation of the cascade on that graph over time. The last column in each table totals the number of points that each team received to make it easy to see the winner.
- `/graph/:id` is where the students can view a simulation of the cascade on the network graph based on everyone's selected seed nodes.

There are three (3) routes to serve the JSON data (as `application/json` MIME type) necessary to make the visualization for the epidemic simulation. These are hit by the javascript `public/js/render-graph.js` that runs client-side to draw the svg and animate it.

- `/api/v1/graph/:id/structure` returns the graph file itself, i.e. the connectivity of the nodes. Note that this is the same file that is served to the user with the `/submit/:id/download` route.
- `/api/v1/graph/:id/layout` returns the positions, i.e. (x, y)-coordinates of each node in the graph. These were produced using d3.js with the `worker.js` file (see "Generating layouts" for more details). Note that the layouts for the graphs are precomputed because rendering each step of the physics simulation is infeasible for large graphs (in the browser).
- `/api/v1/graph/:id/model` returns the differential of changes in node colors as produced by the simulation program. See the "Output format" subsection in "Simulation" section.

*d3.js*
d3 is a fantastic graphing library that is useful for creating all sorts of visualizations. We use it to create the layouts for the different graph files and animate the epidemic on them.

Note: This is hopelessly false, d3 is a piece of crap. ☺ -- Nikita

## 2 Simulation

### Introduction

The simulation portion of Pandemaniac is the part that takes every teams' seed nodes and runs the epidemic model on them until there is no longer any change in the node configuration. This needs to be run every day after the upload period has ended for that day. The simulation code will automatically upload the results to the MongoDB database after it is run so results can be seen immediately on the website.

**Files**
All files are located in `<shared folder>/pandemaniac/code/pandemaniac-modelsim`. There is also a git repository located at https://github.com/visemet/pandemaniac-modelsim, and it would be great if you (the TAs) could push to this repository.

- `models/` - The different epidemic models. Currently, `majority_colored` is used (which means a node will change to a color that a majority of its colored neighbors have). To change this, the two calls to `do_main` in `run.py` must be changed.

  Update: Now, to change this, just change the EPIDEMIC_MODEL parameter in CONFIG.py.

- `CONFIG.py` - Contains all of the configuration parameters. Should be self-explanatory. **Please note that the graph names and configurations for the `DAYS` field must match that for the web application.**
- `main.py` - Handles input and output into the simulation, such as reading in the seeds nodes for each team and uploading the results of a simulation to the database.
- `match.py` - Randomly matches teams with each other. This does not need to be modified.
- `run.py` - Does a run for a single day, using the details in the `DAYS` field in the `CONFIG.py` file. This may need to be changed depending on how TA teams are handled. The filler teams (more details in **Usage**) are created here.
- `sim.py` - The simulation code which is provided for students. This should match how `simulation.py` works. If the epidemic model changes (i.e. it is no longer `majority_colored`), then the code in `sim.py` must change too.
- `simulation.py` - Runs the epidemic simulation. This doesn't need to be changed. If the epidemic model is to be changed, you simply need to add a new file to the `models/` folder and add a new `if` statement in the `__init__` function.

**Setup and installation**
The following tools are needed for the simulation, and exist on the CS cluster already:
- Python 2.7 (32-bit)
- pymongo 2.6.2, used for Python-MongoDB interface. Install with
  ```
  python setup.py install --user
  ```
- MongoDB 2.4.6

The `DAYS` field in `CONFIG.py` must be adjusted accordingly, depending on how the competition is run this year.

**Usage**
To run the simulation, you have to specify the day (so it knows what graphs to run the epidemics on), and the team names:

```
python run.py --day [day number OR round name] --teams [list of teams]
python run.py --day 5 --teams team1 team2 ...
python run.py --day round3 --teams a b z x y

Update: if you just run with –-team s all it’ll just run all teams for a
day/round automatically. Like:
python run.py –-day 2 –-teams all.
```

*Matchings and filler teams*
As it is currently structured, graphs have a specified number of teams competing on it (2, 4, 8, or 16 teams). Teams provided in the `--teams` argument will be randomly matched. However, if there aren't enough teams, then filler teams will be created. For example, if there are 14 teams and there is a graph that needs 16 teams, two teams; `filler1` and `filler2`, will be created and randomly matched with the other teams. The filler teams will only select random nodes.

*TA teams*
Currently there are three TA teams: `TA_fewer`, `TA_degree`, `TA_eyeball`, which play 1-vs-1 with every student team on specified graphs. The selected nodes for these TA teams can be found in the `ta-graphs` folder. Unlike normal graphs which specify the number of players per graph in the `DAYS` field, TA graphs will instead have the name of the TA team (which should also be the name of the folder containing their nodes).

Update: We’ve changed from TA_eyeball to TA_more, the extra credit team. Same deal applies.

*Output Format*
The results of the simulation are outputted to three different locations:

1. A file located at `pandemaniac-graphui/private/runs/` (on the server). The file will be of the form `<graph name>-<timestamp>.txt`.
2. MongoDB database. The results are inserted into the runs collection. The `file` field will be the name as specified in output (1).
3. `stdout`. The output will be of the form:
   ```
   Graph: <graph name>  Teams: <list of teams>
   <first place team>, <first place team's rank>
   <second place team>, <second place team's rank>
   ...
   <last place team>, <last place team's rank>
   ```

## 3 Generating datasets

*Getting graphs*
Code for generating datasets can be found in the `<shared folder>/pandemaniac/Generation/tools` folder. Graphs were taken from the Stanford

SNAP repository (http://snap.stanford.edu/snap/), and converted from a list of edges to a JSON file containing nodes as keys and and a list of that node's neighbors as the value.

To convert the original dataset to JSON, run the script `jsonify.py`:

```
    python jsonify.py [in-file [out-file]]
    python jsonify.py soc-Epinions1.txt epinion.json
```

*Generating subgraphs*
Since we want graphs of different sizes, we used the forest fire model to generate subsets of the aforementioned graphs.

Refer to *Sampling from Large Graphs* (Leskovec and Faloutsos) for an explanation of what the forest fire model is, located in the `<shared folder>/pandemaniac/Generation/papers/` folder. The paper also details several other alternatives, but for reasons that hopefully become clear, the process by which the subgraph is generated using the forest fire model is similar to that of an epidemic itself.

Note that the paper refers to forward and backward burning probabilities; however, since we are dealing with undirected graphs, these are equivalent. Denote this probability as `p`.

As explained in the doc comment of the `forest_fire.py` code itself, the model starts by uniform-randomly selecting a node in the graph. Call this node `v`. Assuming that we have yet to reach our desired subgraph size, we then draw a value `x` from a geometric distribution with mean `1 - p`. This represents the number of vertices incident to vertex `v` that have not been visited to add to the subgraph. Repeat this process for these vertices `w1`, `w2`, …, `wx` until enough nodes are burned. If the "fire" dies out, then we start again by choosing a node uniform-randomly from the graph that has not been visited.

To generate a subset of a graph, run the `forest_fire.py` script:

```
    python forest_fire.py -n [# of nodes] -p [probability param] [in [out]]
    python forest_fire.py file.json file-subset.json -n 1000 -p 0.6
```

The probability parameter can be arbitrarily chosen, but the paper recommends using a burning probability of at least 0.6 for best performance (measured by similarity to the original graph).

*Picking subgraphs to actually use*
The datasets we chose had several thousand nodes, and we chose subsets between 100 to 5000 nodes. For example, in order to generate 100-node subsets from the original 8000-node graph, we generated 1000 subsets. We then ran different strategies on them (for more information, read the section "Generating strategies") and pitted them against each other, and selected the graphs in which a purely random strategy did *not* win. These were done with the scripts `graph-test.py` and `main-test.py`.

*Re-indexing*
After generating the graph, it must be re-indexed so the nodes are numbered from `0` to the

`<graph size> - 1` (for the students' convenience). To do so, run the `reindex.py` script:

```
      python reindex.py [in [out]]
      python reindex.py graph.json graph-reindexed.json
```

## 4 Generating strategies

We came up with several possible strategies that the students might attempt. We used these for tests as well as for determining which were good subset graphs to use (see the section "Generating datasets" above).

| **Strategy (Centrality Measure)** | **Description** |
|---|---|
| degree | Pick the nodes with the highest degree. |
| r-degree | Rank the nodes by degree. Then pick nodes from this ranked list "randomly" following a normal distribution. |
| closeness<br>betweenness<br>clustering | Pick the nodes with the largest closeness / betweenness / clustering measure. |
| r-closeness<br>r-betweenness<br>r-clustering | Rank the nodes by closeness / betweenness / clustering. Then pick nodes from this ranked list "randomly" following a normal distribution. |
| d-clustering | Finds the nodes with highest degree and clustering, then randomly chooses from those nodes. |
| b-clustering | If trying to find N seed nodes, then finds the top N * 4 nodes using their clustering coefficient. Then, of those N * 4 nodes, ranks then by betweenness and finds the top nodes. |
| random | Randomly selects nodes. |

*TA Strategies*

- TA-eyeball: One of the TAs actually looked at the graphs and selected nodes that looked good. He determined which ones were "good" by looking at how clustered they are, how close they are to other nodes, etc. This was all done by eyeball, i.e. without computing anything.
- TA-fewer: Since TA graphs were pre-determined, the TA-fewer team chose nodes for a graph by running all strategies against each other, finding which strategies won, then pitting those winning strategies against each other again. This continues until there is no longer a clear winner. Afterwards, the TA-fewer team randomly selects nodes from the remaining strategies' seed nodes.

- TA-degree: This TA team simply ranks all of the nodes by degree and picks the highest ones. As it so happens, this tended to be one of the more difficult teams to beat, although the students knew exactly what we were going to do. One thought that came up in discussions after the competition would be to do different "levels" of the degree strategy, where the level number indicates how many *more* seed nodes the degree strategy gets than the students.
- TA-more: (seems to be new since this started), so not actually sure how these nodes were chosen. Extra credit team.

## 5 Generating layouts

Update: Read this, then look at “How to not break the visualizer”.

Last year, we had difficulty installing the `d3` module as a package for NodeJS. This may have been due to that the `node` and `npm` commands were not accessible on the global `PATH`, or an incompatibility with the version of Python installed on the CS cluster. Details are unclear.

Instead, you should clone the visualization repository via

```
    git clone https://github.com/visemet/pandemaniac-graphui.git
```

and then run ``npm install``. Note that this requires you to be running NodeJS on your machine (not very difficult to set up).

The `worker.js` script is responsible for generating the layout (the x- and y-coordinates of each node) for a particular graph. We do this by running the physics model (aka force layout) that d3 provides until it converges. Note that since we do not bother rendering the SVG, this process is not *that* time-consuming.

A simple way to regenerate the layout for all graphs is to do

```
    find private/graphs -name "*.json" | xargs -n 1 node worker.js
```

Note that the resulting layouts are automatically put into the `private/layouts/` folder. You can then copy them (`scp`) to the corresponding folder in the CS cluster.

## 6 Server setup

All of the code for the server is already on the CS cluster under the `/cs/courses/cs144/pandemaniac/` and `/cs/courses/cs144/pandemaniac/pandemaniac-graphui/` folders. There are already useful scripts so you probably will not have to interact with the `pandemaniac-graphui/` subdirectory directly (well… not *that* much).

Update: all the code is on the AWS instance. You should have the PEM file and everything is set up; you shouldn’t need to do anything other than run init…to reset everything.

*What the different scripts do*
The **`copy-ta-graphs.sh`** script will copy the contents of `pandemaniac/ta-graphs/` folder to the `pandemaniac-graphui/private/uploads/` folder (as though they were a team that had uploaded online through the web application). This is necessary because the number of seed nodes per graph is enforced by the web application and is (supposedly) constant across all teams. Note that this script is implicitly called by the `init-pandemaniac.sh` script.

The **`init-pandemaniac.sh`** script will remove the contents of the `pandemaniac-graphui/private/runs/` directory as well as the contents of the `pandemaniac-graphui/private/uploads/` directory. The seed nodes of the TA strategies are then copied back into the `uploads/` directory via the `copy-ta-graphs.sh` script. The script then clears out the different collections of the database, and then loads the graph configurations as defined by `pandemaniac-graphui/setup/setup-graphs.js`. **IMPORTANT**: This operation requires that the database actually be running. Finally, the indices are added to the various collections as defined by `pandemaniac-graphui/setup/ensure-indices.js`. Note that it is *unlikely* you will need to this file, unless you change the schema and the core application.

The **`make-db-backup.sh`** script makes a copy of all the collections in the database and puts it in a folder within `pandemaniac/db-backups/` named as the Unix timestamp of the operation. Note that we did not write a restore script. See this article for more details. **IMPORTANT**: This script requires that the database actually be running… which it should be since you would be doing this while the application is running (without taking it down).

The role of the remaining scripts `start-pandemaniac.sh`, `start-pandemaniac-nohup.sh`, and `stop-pandemaniac.sh` are fairly self-explanatory. The `redis-server`, `mongod`, and `node` processes are all started in the background, with their pids saved to a file. This is done so that we can terminate the processes when we take down the web application (without having to use the `pidof` command). The reason for using the nohup command is so that exiting the ssh session does not cause the web application to terminate (which it otherwise would, unless you use tmux or something equivalent).

*Setting up - for the first time*

- If you have not done so, go ahead and first generate the different graphs to use in the competition, as well as the TA strategies. Refer to sections "Generating datasets" and "Generating strategies".
- Update the dates at the top of each of the `pandemaniac-graphui/setup/setup-graph-xx.js` files to change when the different graphs become available. Note that the descriptions for the graphs (i.e. the text itself) can be changed in the `pandemaniac-graphui/setup/setup-graphs.js` file.
- Start up the web application by running `` `sh start-pandemaniac-nohup.sh` `` and run the initialization script, `` `sh init-pandemaniac.sh` ``. If at any point you want to take the

web application offline, just execute `` `sh stop-pandemaniac.sh` ``. It really is that simple!

RECOMMENDED: Have whoever is running the simulation perform a backup of the database first in case anything goes haywire.

IMPORTANT: Make sure that you use the `*.conf` (i.e. `redis.conf`, `mongodb.conf`) files (done so automatically in the scripts provided) because these will ensure that the Redis server and the MongoDB database are accessible only from the local machine (which can only be done by members of the `cs144` group on the CS cluster). You will also need to have Adam to add you to this usergroup.

Update: I don't think this is relevant any longer.

## 7 How to not break the visualizer

Depending on when you TA this class you may or may not remember that the visualizer was broken for a couple of years. This is because d3 is crap. According to the interwebs d3 and NodeJS just have a lot of problems working together; I tried rolling back versions but it didn't really help. The problem seems to be that NodeJS was making calls to d3 to create objects, but these were either unimplemented or deprecated. These calls were throwing errors because whatever the developers had in mind for future implementation meant that they were required to have certain arguments; providing dummy arguments seems to have fixed the problem. I wasn't smart enough to document as I went, but if you run into problems email me (Nikita); at this point I've read enough about d3 that I should be able to help you.

So to use the visualizer

```
npm install
then:
find private/graphs -name "*.json" | xargs -n 1 node worker.js
```

Basically as described above. Hopefully you shouldn't have a problem. Before making any updates to the d3 package, I would make sure you have the current code saved somewhere, so if something starts breaking you can just use the current code – *the current code is NOT a version of the d3 package*, but the most recent version of the d3 package with my edits.

## Appendix A - enforcing time constraint

*Introduction*

First and foremost, note that client-side validation is more about convenience for the user than it is about providing any kind of safety about the data received by the server. Enforcing the submission time limit is but one example of an action that has to be validated server-side. We do this by viewing the different operations as a state machine as follows.

*State Machine*

1. The user sends a request to download a particular graph.
2. If the current time:
    a. Falls between the `start` and `end` times of the graph, then an entry of the form `{ team: <team_name>, graph: <graph_name> }` (IMPORTANT: with the `at` field omitted) is added to the <u>attempts</u> collection if such an entry does not already exist in the collection. A key is also added (if one does not exist) in Redis that represents this particular (`team`, `graph`) combination and is set to expire in `timeout` seconds.
    b. Is greater than the `end` time of the graph, then nothing is recorded in the database.
    c. (somehow) is before the `start` time of the graph, an error status of `400` is returned because the user has made a bad request.

   In both cases 2(a) and 2(b), the graph file is served to the user.
3. Consider whether the student (successfully) submits for the graph (a) zero, (b) one, or (c) more than one times before the key in Redis expires.
    a. The key in Redis will then expire, and the corresponding entry in the <u>attempts</u> collection will still not have an `at` field.
    b. The entry in the <u>attempts</u> collection will have an `at` field with a value of a singleton list containing the timestamp of when the user made their submission.
    c. The entry in the <u>attempts</u> collection is updated and the timestamp of their submission is appended to the `at` field list.

*Error situations*

- If the user (somehow) tries to upload a submission for a graph that has expired, i.e. no key in Redis for (`team`, `graph`) combination, then similar to 2(c), an error status of `400` is returned because the user has made a bad request.
- If the user makes an invalid submission (e.g. too many seed nodes), then the user is redirected back to the graph submissions page with the corresponding error message. Nothing in the database is modified.

*Conclusion*

- Ongoing attempts for a graph are determined by the presence of the key for the (`team`, `graph`) combination in Redis.
- Successful attempts for a graph are determined by the presence of the `at` field in the corresponding entry of the <u>attempts</u> collection.