Web-Based Linear Visualization of Assembly Data

June 22, 2016

1 Project Goals

What do we want the visualization tool to be like? What functionality do we want it to have?

- Provide a web-based platform for analyzing this data.
  - I don't think any current visualization tools for assembly data are web-based, so this would be something new there.
  - By providing sample datasets with the web tool for users to view, users could interactively gain an understanding of the tool's purpose (and, e.g., the challenges of the "finishing" step of assembly) even without having access to assembly data.
- "Do more" for the user than other assembly tools (Bandage, ABySS-Explorer, etc.).
  - Provide an (ideally linear) layout of the longest path through the sequences.
    - As opposed to something like Bandage, where every contig is just displayed on the screen at once.
    - Cytoscape.js' `breadthfirst` layout, which organizes a graph's nodes in an ideally hierarchical structure, seems sort of like what we'd want here. There are plenty of other options, though.
    - One thing Todd suggests: a horizontal scroll-bar (somewhat like a ruler?) on the bottom of the screen that shows the current position in the genome.
    - The natural problem that arises here is that the longest-path problem is NP-Hard. There are ways to mitigate this, though (e.g. only look at a certain region of the graph). Right now, the Python script can detect and sort the connected components of the graph by size, which isn't exactly what we want (a path) but is, hopefully, somewhat useful.
  - Highlight interesting structural patterns in the sequence, e.g. bubbles, frayed ropes, and chains.

    * The Python script can already do this, and this shouldn't be hard to implement again relative to whatever way we represent the graph. The method I'm using takes roughly $O(2n)$ time in the worst case, when each node is checked as both a frayed rope and as a chain (since both structures involve starting nodes with exactly one outgoing edge, so if the current node is neither a frayed rope or a chain we have to check both). I could probably make this code a decent amount more efficient (it's not that pretty), but this isn't really the main bottleneck here.
    * An interesting question here is whether we'd want to "collapse" these patterns into a single node or keep the nodes as is, but group them into a marked cluster representing that group of nodes. I've implemented both behaviors in two versions of the Python script (`collate_double.py` and `collate_clusters.py`), and we can definitely provide users with the ability to choose which method of representation they prefer.

- If we wind up using GraphViz' `xdot` file format for layout, then provide the option for the user to view (and, if they want to, tweak) this code? Many online GraphViz libraries' example tools have this option, e.g. `http://mdaines.github.io/viz.js/`.

- Provide user with ability to highlight and save information (e.g. the sequence of nucleotides, a picture of the contig layout, ...) about a certain portion of the graph. Bandage supports this, and it'd be a nice feature to have.

- Support display of either a single graph (each contig is "merged" with its reverse complement contig, and each contig is therefore directionless and rendered as a square/rectangle/etc. instead of a house/invhouse) or a double graph (the RC of a contig is displayed as a separate node).
  If we have the time to do this, it would also be cool if we could automatically switch to a double graph as the user zooms in on the graph (and to a single graph as the user zooms out). *However*, this might be impractical due to 1) taking a nontrivial amount of time to relayout the graph, therefore preventing the user from zooming in, and 2) the layout resulting from rendering RC nodes as well as "merged" nodes would probably result in the initial "merged" nodes having a different position than they originally had, disorienting the user.

- This is a fairly long-term goal (we'd probably focus on it after much of the work getting a solid base for the tool has been done), but one thing Todd suggested was the ability to load multiple files of the same genome/region produced by different assemblers. This would provide functionality for the user to compare assembly quality (e.g. "ABySS just produced a bunch of disconnected short sequences here, but Velvet produced a longer path") and maybe combine information from the two assemblies.

2 High-Level Tool Structure

I'd expect the tool to be composed of, at minimum:

- Something that **parses assembly files** (we should support multiple types of files, not just Velvet's LastGraph files) and converts them to a graph format (either to feed into GraphViz for laying out, or to feed into whatever graph library we use if we use that for layout). The python script already does this, along with detecting certain features of the graph (e.g. connected components, bubbles/ropes/chains). An added note: we should probably have some **security measures** in place here, to prevent users from doing something silly like trying to inject code through assembly files/filenames.
- Something that **performs layout**: Determining how to display the graph to the user. When looking at graphs with lots of nodes (say, ≥ 1000) this seems to be an intensive operation, but I think drawing the nodes is actually the biggest bottleneck here. Running `dot ecoli_clusters.gv -Txdot > layout.xdot` on my system takes around 0.4 seconds (I profiled it using `time` three times), but running `dot ecoli_clusters.gv -Tpng > layout.png` on my system takes around 4.5 seconds.
  For reference, the E. coli assembly file I'm using to test these things contains 297 contigs (not counting reverse complements; accounting for those, as the python script does, results in a graph of 594 nodes.)
- Something that **draws the graph**. This is going to be dictated by which graph-drawing library we choose: there are a lot of options. As of now, I've looked into vis.js, sigma.js, arbor.js, and cytoscape.js. The specifics of these libraries are discussed in the next section.
- If we use a layout generator that isn't a part of whatever graphing library we choose to use, we'll also need something that **parses layout files** and sends that information to the graph-drawing library. The most likely case (if this ends up happening) is having to parse .xdot files generated by GraphViz, which shouldn't be that bad.
- A **user interface** in which the user can, in addition (if necessary) to the interactivity provided by the graph-drawing library:
  - Collapse/highlight certain types of nodes
  - Move nodes
  - Zoom in/out
  - Select certain nodes/groups of nodes

  This is basically going to involve implementing the features detailed under "do more" in section 1, and implementing those things (and explanations for how to do them/what they do) in the tool.
- A **documentation/help system** detailing how the tool and its features can be used. Especially with a web application, there's no way to guarantee that the users of the tool understand what's going on, so it's probably a good idea to have some guidance available—we can refine this later on, as functionality is implemented.

3 Technical Details

The previous section describes a high-level overview of the components involved in the tool. This section will describe, at a closer level, how those components will interact.

I'm admittedly pretty new to web development, so I'm going to have to learn a lot of that throughout the project. Sorry if some of the concepts here aren't communicated that well!

Ideally, a user would just load the tool's website and be able to interact with it there. The user could easily upload an assembly file, which would then be parsed on the server side, laid out as a graph, and drawn on the screen for the user to interact with.

1. **Parsing assembly files:** We could definitely parse assembly files using the Python script on the server side, I suppose.

2. **Laying out the graph:** From there, we can (depending on our approach) run GraphViz, either on the server side or on the client side (using one of the GraphViz javascript libraries). If we elect not to use GraphViz for layout, then we can use the graph-drawing library to control graph layout (in which case we'd skip this step).

3. **Drawing the graph:** If we elected not to use GraphViz, we can just enter the nodes and their connections into the graph-drawing library at this step. If we did use GraphViz for layout, then we will have to parse its xdot output and send that positioning information to the graph-drawing library at this step.

   All of the drawing libraries I've seen thus far have been in Javascript, and most of them use/support Node.js. Might as well put it here—current candidate graph-drawing libraries:

   - vis.js

     + Relatively fast when fancy physics settings and smooth edges are turned off (loads 5,000 nodes and edges in approximately 8 seconds; 10,000 nodes and edges in approximately 17 seconds!) This is including the costs of laying out the graph, also.

     – I'm not sure if there's a way to use a preset layout in the graph. Even if not, though, there's probably a way to get around that (the library is open source, so I can always change things)

   - sigma.js

     + Seems like a fairly well-known and supported library (probably lots of documentation/etc. available)

     – Dragging nodes is only supported with the `dragNodes` plugin. The plugin also felt strange to use; the nodes I was moving felt like they would go flying off the screen suddenly.

     – Using 10,000 nodes and edges with the dragNodes plugin was very slow; combined with the shakiness of the plugin, the lag from many nodes and edges made the graph feel almost unusable.

- arbor.js

  + Looks fancy?

  – Not a lot of documentation available; there's a tiny introduction, a decent reference page that goes over the library's functionality, and one basic tutorial that someone else made (I found it on google), but beyond that I haven't found/seen much.

  – The last update to the project's website looks to have been in 2011, and the last commit to the project's repository was in 2012. Granted, the project could just be really stable at this point, but I'm not optimistic about how many resources we'd have if we get stuck somewhere (or if the library has a bug that impacts our problem, etc.)

  – It looks like the main focus of the system is force-directed graphs, which is probably not what we want (physics simulations are interesting, but really expensive on big data sets. Also, since we want to display the assembly data linearly, we'd probably end up overwriting this anyway.)

- cytoscape.js

  + Very well supported—lots of documentation, tutorials, etc.

  + Many layout options (including preset, should we decide to use GraphViz for layout).

  + It looks really nice, and interacting with the graph is very smooth, until you reach really high (somewhere between five to ten thousand) nodes. Even then, the graph is still usable.

  + Supports batch drawing operations, which makes atomically adding/modifying nodes easy. For reference: It drew 10,000 nodes and edges in about 6 seconds, and I'm sure there are ways to tweak the library's settings to make this even faster. (However, this is using a random layout, so the cost of laying out the nodes isn't factored in— however, using the cytoscape.js "circle" layout took about the same amount of time as the random layout, so I don't think it'll make that much of a difference.)

If I had to rank these graphing libraries for the purposes of this project, I'd probably say it goes something like 1. cytoscape.js, 2. vis.js, 3. sigma.js, 4. arbor.js. I spent a few days looking into them/testing them, but I probably could've found more out about each.

cytoscape.js, from what I saw, was relatively fast, highly configurable, and well-documented (with a large user-base, also).

vis.js was similar to cytoscape.js, but it had slower loading times for really large graphs. I'm guessing it has a smaller userbase than cytoscape.js, also, but that's kind of a minor difference at this point.

sigma.js seems to focus most on static graphs—not being able to "natively" drag nodes is a pretty significant drawback, at least in my opinion. Dragging issues aside, sigma.js was fairly good with displaying a large amount of nodes/edges: I don't think its performance was better than cytoscape.js (or even vis.js?), though.

arbor.js' website admits the library is focused on force-directed graphs, which isn't what we want from a graph-drawing library for this project. That, plus the sparse documentation and small scope of the project, is a drawback for arbor.js in this context.

One last point re: graph drawing: this stack overflow question (`https://stackoverflow.com/questions/31364329/scalability-of-cytoscape-js`) features a response from (one of?) the developers of cytoscape.js, in which he says that browsers are the limiting factor for drawing really large graphs (so a graphing library can only do so much to optimize things).

4. When the user makes significant changes to the graph (e.g. collapsing all bubbles, ropes, and chains), we may want to **re-layout** the graph. With cytoscape.js this can be done relatively painlessly using `cy.batch()` and related functions; not sure how we'd go about doing that with the other graphing libraries.

5. All the graphing libraries discussed above display the graph in the user's browser—as far as I can tell, they all support using Node.js to do computation on the server-side (although the performance tests I ran were local, on my computer). Whether we want to offload computation to the server- or client-side depends on the graph library we use, I guess, and how much volume we'd estimate having? (Like I said, I don't know a lot about this side of things yet.)