Paul Duchesne (B00332119)
CSCI 4166: Visualization

Project Report: Pathfinder Feat Visualization

Introduction

The goal of this project was to visualize the dependencies required by the thousands of of feats used in Pathfinder, a tabletop roleplaying game similar to D&D. The end goal was to create independent dependency trees for each individual feat individual feats. It should be noted that as the progress progressed, the goals changed from those outlined in the project proposal and update. However, the primary goals outlined in the project update were met. The database used in this project was found online here, and was originally created by scraping the main pathfinder online 'system reference document' (SRD) located here. This project used an HTML framework to run various javascript files using the D3 visualization library. Specifically, the D3-Force framework was heavily used to create a dynamic visualization of nodes connected with links. Furthermore, a functional clone of D3's now unsupported tip library, found here, was used to create dynamic tooltip displays.

Task Breakdowns and Challenges*

1. Formatting Database into CSV File:
Work was required to format the original .xlsl file (Google Sheet) before exporting it as a CSV file. Glaring typos, duplicates, and inconsistent formats were fixed both manually and with temporary scripting in an iterative process.

2. Parsing the Data from CSV File:
Once formatted, the data was read into a javascript file by the native D3 CSV parser after serving the project with a Python based SimpleHTTPServer. This data was locally stored in an array and then parsed using newly created functions to split and sort the data into different categories. This can be seen in *format_csv_data()* within *main.js*. In particular, the following fields fields were tokenized and stored within new fields of the array: 'prerequisites', 'prerequisite_feats', 'prerequisite_skills', 'race_name', and 'type'.

3. Data Visualization with D3-Force & SVG:

By defining a simulation using D3's built in force simulation API, it was possible to create dynamically linked nodes to represent each feat in the dataset. The goal was to create a node dependency tree for each node individually from that particular feat. Each tree consists of all prerequisite nodes linked in sequence, with no aliasing and no unnecessary links, as seen in Figure 1.

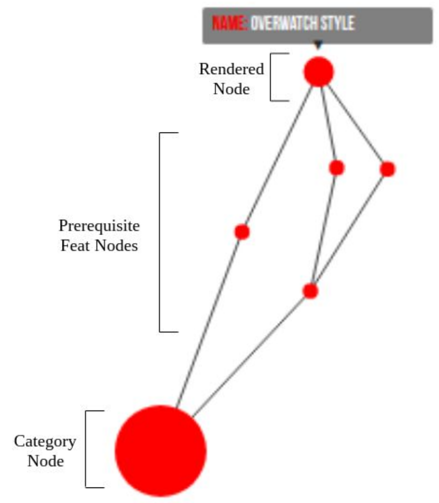

Figure 1: Example Dependency Tree

These trees link down an array of large rooted nodes, each representing one of the eight feat categories. In this array, the central category nodes have a larger radius than the feats and their prerequisite dependency feats. This can be seen in Figure 2.

* *For more information about sections 1 and 2, refer to the midterm report*

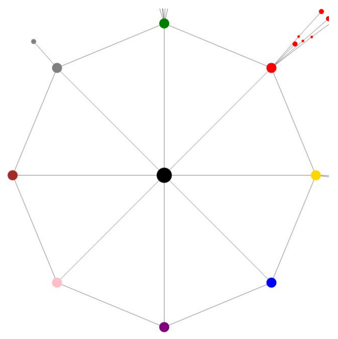

Figure 2: Central Category Nodes

Figure 2 shows that rendered nodes were given a larger radius than their prerequisite feats, a smaller radius than the prerequisite nodes, and a smaller radius than the category nodes.

A large part of development within this section was the creation of a single function, *create_dependencies()*, that could be called on any feat within the input data array and create the required nodes and links of its individual dependency tree. The entirety of *node_tree_functions.js* consists of this function and the functions it requires to perform a single call. This function was a challenge because the dataset used did not differentiate between explicit and implicit prerequisites when listing them. In this case, an implicit prerequisite is one that is not met by through having other necessary requirements. On the other hand, explicit requirements are those aforementioned necessary requirements. This is outlined in Figure 3, which shows a feat dependency mockup for a hypothetical feat named ‘Fancy Swords’. The left side of the image shows the ‘Fancy Swords’ as described in the database, with ‘Fancy Swords’ requiring both ‘Swords’ and ‘Decent Swords’, which in turn requires ‘Swords’. By inspection it is obvious that ‘Fancy Swords’ does not actually require ‘Swords’, it simply requires ‘Decent Swords’, which requires ‘Swords’ to achieve. In this case, ‘Fancy Swords’ is said to explicitly require ‘Decent Swords’ and implicitly require ‘Swords’. Thus the left hand tree can be simplified to the one shown on the right.

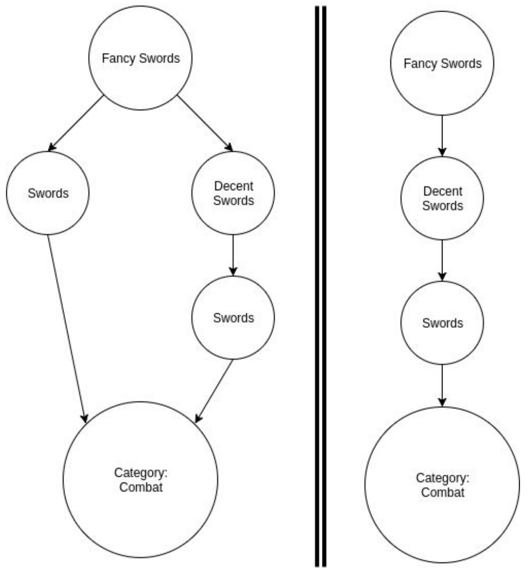

Figure 3: ‘Fancy Swords’ Dependencies

This issue plagued the dataset and multiple functions were required to break the requirements apart and determine what was truly required for each node. Within this process, aliasing had to be avoided at all levels and each sub-prerequisite had to be checked for its own explicit and implicit requirements iteratively to create a proper mapping from the top node to the root category. Other processing was required to store node layer data in order to create proper links with modular link distances set.

The last task for this section was the addition of dynamic tooltips using a version of D3’s now unsupported tip library. While this library is no longer officially supported by D3 v4, a functional copy that has been ported for v4 was used for this project. This allowed tooltips to dynamically display not only the name of a feat, but also the skills, ability scores, races, and base attack bonus (BAB) required to take a feat.

4. Modularly Re-rendering the Visualization:

This task broke down into two smaller tasks. The first of which was modularly rendering the central category nodes and the second was clearing and updating the existing nodes to reattach the new layout.

The first task was relatively simple to achieve and was completed by modularizing the previous implementation into a new function, *initialize_central_nodes*, which has the ability to recreate the central category nodes in any layout, not matter the number of categories used in a given rendering. For example, Figure 4 shows this in action with only 5 feat categories being used.

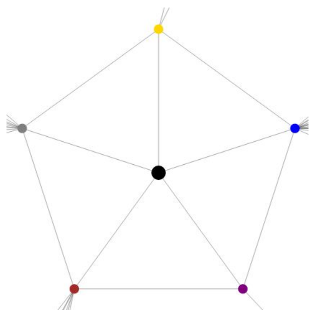

Figure 4: Modular Central Category Nodes

The second task a surprising amount of time due to the complex nature of D3's relation with SVG, the DOM, and Javascript itself. This was further complicated by the somewhat arcane nature of D3 in terms of data binding. This led to some time being required to determine how to remove all node and link elements from the locations: the local javascript storage (nodes, links), the SVG canvas (svg_nodes, svg_links), and the DOM (through d3 selectors). This was eventually resolved through multiple functions within *render_restart.js*, which collectively find and delete all unwanted elements before creating new ones to render. Once this task was complete, it was possible to go back and add 'onClick' events that modularly render individual nodes and their dependencies by clicking on them. It was also chosen to filter by individual categories if a central category node is clicked, or all nodes if the root node is clicked. It should be noted that for these two options, all nodes with zero prerequisite feats were excluded from the rendering to minimize the large number of nodes being created.

5. User Interface for Search Options:

A rudimentary user interface was created in order to allow the user to search and filter the feats within the visualization. This allows them to search by feat name, race, ID, and category. To accomplish this, input data was required within the javascript from the user. Because D3 does not natively support input fields, this was done using SVG's 'foreignObject', which allowed HTML input forms to be placed within the SVG canvas. While this has not been tested with this system, it is a well known that foreignObjects are not supported in Internet Explorer.

Due to the preprocessing done on the input data, searching the data was a relatively easy task that simply involved iterating through each option and narrowing down the results.

6. Render Cache:

A small cache was added to save the parameters of each render, thus allowing the user to iterate backwards through displays. This can be done by clicking the circular button in the upper left portion of the menu. To avoid caching infinitely and wasting memory, the cache was given a maximum depth of eight.

User Interface Notes

For a detailed description of the network from a user's perspective, please refer to the README.txt found in the main directory of the project. This further details the different shapes present in the visualization and their meanings, as well as a detailed description of the relevant menu options available.

Limitations & Future Work

A disproportionate amount of time was spent formatting the data and creating proper dependency node layouts. Unfortunately, this resulted in relatively few features for the final visualization, with only simple transitions between nodes, a simple user interface for searching and filtering, and a cache function.

It should be noted that this project ended in a state in which new features are relatively easy to add. For example, with the existing framework in place, the cache took ~20 lines to implement (see *svg_initialize.js*). As with most projects, the hardest part was getting a working framework that could then be built from to create more interesting and elaborate features. Below is a list of features that could be added if this project was pursued further. Some of these provide solutions to limitations within the visualization, while others are features that were not implemented.

1. Improved Filtering:
Currently, filtering can only be done by string, race, ID, and feat types. Ideally, this would also include skills, base attack bonus, and a description. Furthermore, users could be given the option to filter either inclusively or exclusively for each category. For example, a user could filter inclusively with "feats that a human can use", or exclusively with "feats that only a human can use". Currently, all filtering is done inclusively.

2. Zooming:
Currently, the relative size of nodes within the system is an issue. It is incredibly difficult to read the tooltips without manually zooming in on the portion of the code. Support exists within D3 to allow dynamic zooming with different filters and effects and any of these would be a great help with aiding the user.

3. Dynamic Legend:
Another issue with the system is that the representations are not intuitive. To use the system, the user must first read the manual (README.txt). To improve this, a dynamic legend could be added with tooltips to give quick explanations about how the system works. Example nodes and their relative sizes could be displayed for an intuitive understanding of the design layout.

4. Resource Reduction:
Despite efforts being made to reduce resource usage, this project still uses far too many resources when rendering a large fraction of the dataset. Possible improvements include:

a) Removing unnecessary portions of the csv data array (and therefore data array).
b) Performing more preprocessing to minimize runtime computations.
c) Provide the option to turn off the dynamic simulation and provide a static display for rendering large numbers of nodes.

Conclusion

In the end, this project met the overarching goals that were laid out in previous reports. The dataset was properly read, parsed, and dependency trees were created for each individual node. Plus, interactive filtering was added to allow user interaction and processing. Overall, this was an interesting way to be introduced to Javascript and D3, with some portions of HTML, CSS, and SVG thrown in for good measure.