Date: May 22nd, 2019
Author: Garret Rieger
Shared Publicly
Revision 2

Introduction

This doc explores the design of a framework to analyze several different approaches to progressive font enrichment (PFE) in order to measure their performance relative to each other.

Goals of the Analysis

- Quantify the network performance (defined in the next section) of each PFE solution.
- For this analysis we're primarily concerned with the network cost for each solution. It does not attempt to measure the client side processing and rendering cost associated with each solution. Nor will it take into account server processing costs for each solution.
- Allow the various solutions to be compared to each other and the current state of the art in font transfer.
- Where possible the analysis should be representative of expected real world usage.
- The results of this analysis will be shared with the public w3c fonts working group, so the analysis framework implementation should be open source so that the results are reproducible outside of Google.

Success Criteria for a Solution

An ideal PFE system:

- Retains all layout rules found in the source font which are needed for the codepoints present on a particular page.
- Loads font data fairly, that is for each page view the amount of font data loaded is related to the number of new codepoints on that page.
- Minimizes the user perceptible delay caused by font loading.
- Has good network performance, that is it minimizes:
    - First, the worst case network delay caused by font loading for clients when viewing a sequence of pages.
    - Second, the total number of bytes transferred (both requests and responses) caused by font loading for clients when viewing a sequence of pages.
    - Third, the total number of network requests caused by font loading for clients when viewing a sequence of pages.
- Network delay is the total time taken starting from the start of the first font related network request to the end of the last font related network request for a single page view.

    - It should take into account parallel network requests (thanks to HTTP2) and varying network conditions where possible.

Input Data to the Analysis

The input to the analysis is a set of **page view sequences** per **font family.**

Font Family:

- Associated font file (e.g. Roboto-Regular.ttf).
- A set of N **page view sequences**.

Page view sequence:

- A set of 1 to at most K **page views.**

Page view:

- A set of unique unicode codepoints (which would be needed to render all of the content for that page view).
- The set of codepoints are just those characters from a page which would have been rendered in the font family this page view is for.
- The glyph set that is needed to render these codepoints in the associated font. Glyph ID's are those from the associated font.

Note on an Alternate Approach

A variation on the proposed approach would be to have a page view contain a map of one or more **family →Unicode code points**. This would more closely model real world usage where a page typically has multiple fonts in use. However, font loads within a page occur largely independent of each other, so it may not be necessary to model multiple font loads within a page view to get accurate results.

Since it's easy to transform input data into either form the analysis code be run using both approaches. If results are similar then the first method could be used as it's simpler.

Acquiring Input Data

This is probably the most challenging problem currently faced in doing the analysis. Ideally, for the most realistic possible simulation page view sequences should be generated based on observed real user page walks. However, it may be difficult to find a large corpus of data containing user page walks. So we might need to instead to generate synthetic page walks across on real web pages. This will only consider the static content on pages and will not account for dynamically added or user generated content on pages.

Analysis

Overview

- For each input **Font Family:**
    - For each individual page view under each page view sequence compute:
        - The network delay (defined in the next section).
        - The total bytes transferred (both requests and responses).
        - The total number of network requests.
        - Note: data caching should be taken into account within each page view sequence (ie. subsequent page views in a sequence can re-use data loaded by previous views)
    - From the data gathered in the previous step we can compute:
        - Distribution of network delays across all page views.
        - Distribution of bytes transferred across all page views sequences.
        - Distribution of number of network requests across all page views.
    - The distributions will help give us a better understanding of how the various solutions perform, but the actual comparison will be done using a cost function and comparing the total cost associated with each solution.

Proposed Cost Function

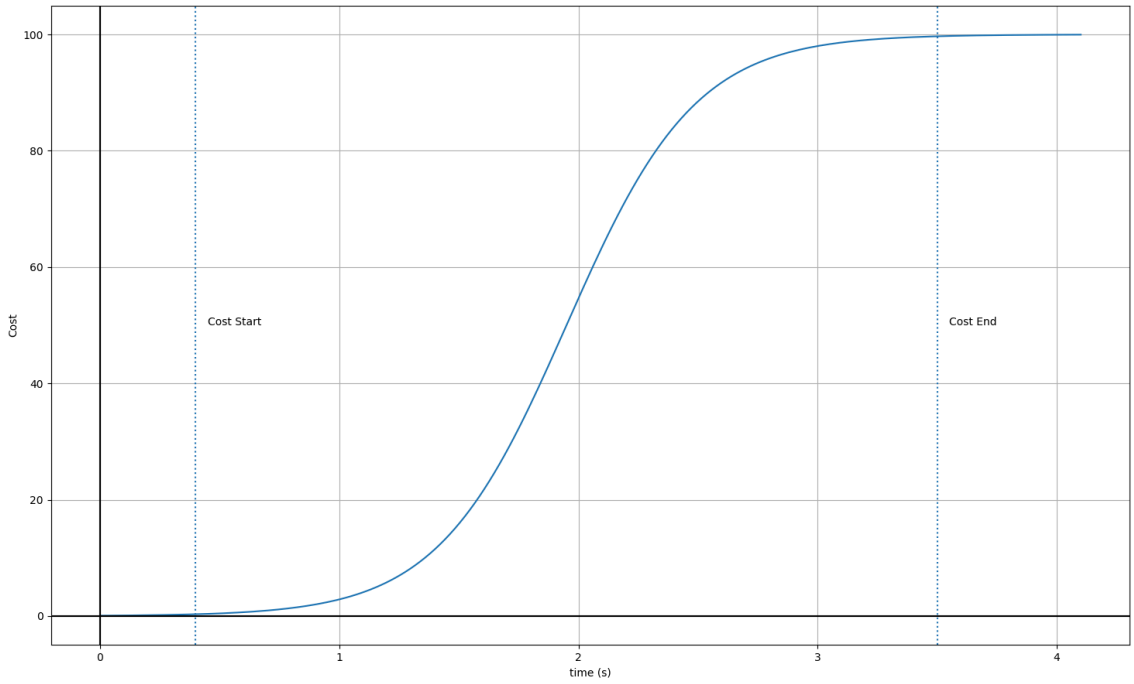
First a set of assumptions this function is based on:

- We primarily care about user experience. Ie. how fast they perceive font loads. Ultimately we want to reduce/eliminate user perceptible delays caused by waiting for font loads.
    - On a given page view there's a window where the rest of the page is loading. If the font loads within this window then there is no user perceptible delay for the font load.
- The negative effect of a loading delay is not linear. (eg. if the user waits too long they may give up on a page, or the browser may decide to fall back to a system font).
- There is a maximum negative effect from loading delay. Once the delay exceeds some amount most browsers will give up on the request and proceed with rendering without the font. So additional cost should not be incurred past this point. **[Added Jan. 2020]**
- Total number of bytes transferred is secondary, but still important. In some cases users may literally be paying per byte transferred (on a metered connection).
    - The smallest number of bytes we can transfer is the "Optimal Subset" which is a subset that contains exactly the characters needed to render all codepoints present in a sequence of pages. It is not possible to transfer less than this amount without degrading the typographic experience.

Then:

- Given the following parameters:
  - T_m is the time at which maximum cost is incurred.
  - T_z is the time before which no cost should be incurred.
  - M is the maximum cost value.
- The cost of network delay t in seconds can then be modeled as:

$$c(t) = \frac{M}{1 + e^{\frac{-11.5}{T_m - T_z} \bullet (t - \frac{T_z}{2} - \frac{T_m}{2})}}$$



- The total cost for a PFE solution is then $\Sigma\, c(t)$ for all the individual network delays over a collection of page views.

- The bytes transferred efficiency is:

$$E(b) = \frac{O}{b}$$

Here O is the optimal subset size.

A good PFE solution should have both network delay cost and bytes transferred efficiency that is better than the current state of the art transfer methods. Between proposed methods for which this is true we would then pick the one with the lowest network delay cost.

Computing Network Delay

- Input to computing network delay is a font family, a set of codepoints, previously loaded codepoints from previous page views, and a progressive font enrichment method.
- Using the above create a directed acyclic graph (DAG) of network requests needed to provide fontdata for rendering the input codepoints given the prior state using the specified transfer method.
    - Nodes in the graph represent individual requests. Each node should have an associated number of bytes transferred.
    - Edges point to requests that are dependent on the source node.
- Weight each node with an estimated network delay =
    - RTT + node.bytes_transferred * delay_per_byte
- Finally total estimated network delay is the path through the graph with the highest total cost.
- It will likely be beneficial to redo the analysis with several different network configurations (latency, and delay per byte) to represent some typical network configurations. For example 3G, 4G, and a typical desktop connection.

Progressive Font Enrichment Function

For each transfer solution we need a function:

- Input is set of codepoints previously loaded, set of codepoints needed, font family.
- Output: DAG of network requests needed to provide font data for the set of codepoints requested. Each node represents a request and should have the bytes transferred (request and response) associated with it.

Proposed Functions

The following functions should be created for the analysis:

- Full Font Files
    - Always send the full font file and then cache it for subsequent views.

- Language based subsets. (Implemented by Google).
    - Send one or more language based subsets based on codepoints needed. Cache subsets/slices for future page views in a sequence.
- Patch and subset. (implemented by Google).
    - Based on Google's current proposal. Compute a subset at the set of codepoints the user currently has and a subset at the set they want. Create a binary diff between the two and send that to the client. There are two variants:
        - with retain gids set (on the subsetter).
        - Without retain gids set (on the subsetter).
- Adobe's existing dynamic subsetting solution.
    - If Adobe is able to provide data to us about page view sequences and associated byte transfer costs from their service we can include that in our analysis.
- HTTP Range transfers (to be provided by Apple).
    - Request pieces of the source font using HTTP range requests.
    - Requires a utility for rewriting a font file to optimize HTTP range requests needed.
- Optimal Transfer (all up front):
    - Assumes we have perfect knowledge ahead of time of all codepoints that will be needed for a page view sequence and cut a subset of the font to that specific set of codepoints. Load in on the first page view.
- Optimal Transfer (accrued):
    - At each page view compute the exact subset that would be needed to cover all of the codepoints on this particular view and any previous views. The request bytes needed for that view is then the difference between the subset size for this view and that from the previous view.

Font Families to Test With

The test corpus should cover a wide range of different fonts and scripts. Some categories that may be of interest:

- Fonts using TrueType, CFF, and CFF2 outlines.
- Fonts covering a wide variety of scripts. Including complex scripts (eg. Arabic and Indic scripts) and scripts with large glyph counts (eg. Chinese, Japanese, and Korean).
- Emoji, icon, and colour fonts.
- Variable and Non Variable Fonts.
- TTC (Font Collections)

The google fonts collection is a large and freely available set of fonts that can be used as a starting point to create a corpus of test fonts. However, it does not provide full coverage of the categories listed above.