Visualización interactiva del grafo normativo argentino a escala: de un régimen de colapso a un régimen de costo acotado

Precómputo del layout de fuerzas, nivel de detalle semántico con umbral invariante de escala, recorte de segmentos por Liang–Barsky, render híbrido canvas–SVG y caché con coalescencia de solicitudes

Ariel Eitner

Observatorio de Leyes — observatorio-de-leyes-front.vercel.app · Buenos Aires, Argentina · Versión 1.0, agosto de 2026

RESUMEN

El Mapa Legal del Observatorio de Leyes visualiza el corpus normativo argentino como un grafo dirigido y ponderado $G = (V, E, w)$ cuyos vértices son normas (leyes, códigos, constituciones, tratados, decretos de necesidad y urgencia) y cuyas aristas son relaciones jurídicas tipadas (deroga, modifica, reglamenta, complementa, etc.). Entre julio y agosto de 2026 el corpus creció de 1.520 a 8.034 vértices (×5,3) y la implementación original —simulación de fuerzas en el cliente y render retenido íntegramente en SVG— colapsó: 3 cuadros por segundo (FPS) en interacción con 3.702 vértices, 50,1 s de bloqueo del hilo principal en la variante 3D, y 75–96 s de latencia por solicitud del endpoint cuando el corpus superó el límite de 2 MB del caché de datos de la plataforma. Este trabajo documenta el diagnóstico experimental que aisló el costo dominante (la *rasterización de aristas SVG*, con un costo marginal medido de ≈59,8 µs por arista y por cuadro, $R^2 = 0{,}97$; no el tamaño del DOM), y las seis transformaciones que llevaron el sistema a un régimen en el que el costo por cuadro y por solicitud queda *acotado por constantes independientes del tamaño del grafo*: (i) precómputo del layout de fuerzas en el servidor (Barnes–Hut, 240 iteraciones, una vez por corpus), (ii) ponderación semántica de aristas w: $T \to \{1,\dots,10\}$, (iii) nivel de detalle con umbral *relativo al encuadre*, invariante ante el tamaño del viewport y del corpus, con presupuesto fijo de aristas débiles, (iv) recorte exacto de segmentos por Liang–Barsky con pre-test de caja envolvente, (v) render híbrido con aristas en canvas agrupadas por estilo (llamadas de dibujo $O(S)$, $S \le 7$) y vértices en SVG para conservar el hit-testing, y (vi) caché en memoria de proceso con coalescencia de solicitudes por promesa compartida y compresión Brotli en el codo de su curva calidad–CPU. Resultados medidos: 3 → 32,7 FPS en 2D (×10,9); 50.123 → ≈230 ms de bloqueo en 3D (×218); latencia del endpoint 75–96 s → 0,35 s en régimen (×215–275); transferencia 2.516 → 372 kB (×6,8). Se presenta el modelo de costo, el análisis asintótico de cada transformación y las mediciones que lo validan, junto con las limitaciones vigentes.

Palabras clave: visualización de grafos · layout dirigido por fuerzas · Barnes–Hut · nivel de detalle (LOD) · recorte de líneas · Liang–Barsky · SVG · canvas · caché · coalescencia de solicitudes · Brotli · legislación argentina · legal tech

1 Introducción

El ordenamiento jurídico argentino no es una lista: es una red. Una ley deroga a otra, un decreto la reglamenta, un tratado se incorpora con jerarquía constitucional, un DNU modifica ochenta normas de un saque. Esa estructura relacional es invisible en los repositorios oficiales, que presentan cada norma como un documento aislado. El *Mapa Legal* (`/herramientas/mapa-legal`) la hace visible: el usuario ve la Constitución Nacional al centro, los "barrios" temáticos alrededor, y puede seguir con el dedo la cadena de

derogaciones de cualquier régimen. Es, hasta donde sabemos, la única visualización interactiva y pública del grafo normativo argentino completo.

El problema de ingeniería es que un grafo así crece más rápido que la capacidad de un navegador para dibujarlo. Entre julio y agosto de 2026 el corpus pasó de 1.520 a 8.034 normas, y la implementación original — correcta a escala chica — entró en un régimen de colapso: la interacción cayó a 3 FPS, la variante 3D congelaba la pestaña durante 50 segundos y el endpoint que sirve el grafo pasó a tardar más que el timeout del propio cliente. Este artículo documenta cómo se diagnosticó y se resolvió, con dos objetivos: (a) dejar el registro matemático y experimental de las decisiones, para que puedan auditarse y estudiarse, y (b) ofrecer un caso replicable para otros proyectos de datos públicos que enfrenten el mismo muro: *la escala del Estado no cabe en un render ingenuo*.

La tesis central es simple de enunciar: ninguna de las optimizaciones consistió en "hacer lo mismo más rápido". Todas consistieron en **cambiar la clase de complejidad de una operación** — de proporcional al tamaño del grafo a constante o amortizada — o en **mover el momento en que se paga el costo** (del cuadro a la interacción, de la visita al corpus, de la solicitud al despliegue). La sección 7 consolida ese argumento en una tabla de complejidades antes/después.

2 El objeto: definición formal y escala

Sea $G = (V, E, \tau, w)$ un grafo dirigido donde V es el conjunto de normas vigentes o derogadas del corpus, $E \subseteq V \times V$ el conjunto de relaciones curadas, $\tau: E \to T$ una función de tipado sobre el conjunto de tipos jurídicos T, y $w: T \to \{1, \ldots, 10\}$ una ponderación semántica (§5.1). Cada vértice lleva además atributos (categoría temática, cantidad de artículos, jerarquía kelseniana) que gobiernan su radio, su color y su estrato en los distintos layouts.

Tabla 1. Escala del grafo en tres momentos. Las cifras de agosto-19 corresponden al corpus local completo, medidas sobre el payload servido (`n` = |V|, `m` = |E|).

MOMENTO	N = |V|	M = |E|	ELEMENTOS SVG	PAYLOAD (CRUDO)
jul-2026 (línea de base)	1.520	2.609	≈5.600	—
14-ago-2026 (diagnóstico)	3.702	7.006	18.112	1,38 MB
19-ago-2026 (corpus local)	8.034	9.137	11.102*	2,52 MB

* Tras mover las aristas a canvas (§5.4) los elementos SVG son solo vértices, etiquetas y grupos.

La distribución de grado es fuertemente heterogénea, compatible con una red libre de escala [7]: grado medio 2,27, mediana 2, máximo 883 (la Constitución Nacional y los grandes códigos actúan como *hubs*). Esta asimetría es central para el diseño: los hubs generan aristas que atraviesan el mapa entero (lo que invalida el culling por caja envolvente, §5.3) y a la vez permiten un nivel de detalle por centralidad (§5.2). La distribución de pesos del corpus actual: w=2 (RELACIONADA): 5.929 aristas (64,9 %); w=8 (MODIFICA): 1.647; w=5 (COMPLEMENTA, SUPLETORIA, ESPECIALIZA): 1.249; w=10 (DEROGA, DEROGA_PARCIALMENTE): 190; w=7 (REGLAMENTA, IMPLEMENTA): 104; w=9 (VETA): 16; resto: 2.

3 Arquitectura original y modo de falla

La implementación de julio de 2026 seguía el patrón canónico de D3 [6]:

1. **Layout en el cliente, por visita.** Al montar el componente se ejecutaban 240 iteraciones (*ticks*) síncronas de una simulación de fuerzas (`d3-force`): atracción por resorte en las aristas, repulsión de muchos cuerpos, colisión por radio y anclas de categoría sobre un anillo. Cada visitante pagaba el layout completo.
2. **Render retenido 100 % SVG.** Cada vértice, cada arista y cada etiqueta era un elemento del DOM (`<circle>`, `<line>`, `<text>`): 18.112 elementos a la escala del diagnóstico.
3. **Reconstrucción total al filtrar.** Cambiar un filtro de categoría ejecutaba `selectAll("*").remove()` y reconstruía el árbol completo: $O(n + m)$ de manipulación de DOM por click.
4. **Sin nivel de detalle efectivo.** Existía un mecanismo de LOD ("ocultar aristas débiles al alejar"), pero estaba *muerto por un bug de datos*: el backend enviaba `weight: 10` fijo para todas las aristas, de modo que el predicado `isWeakEdge = (tipo ∈ {RELACIONADA, referencia}) ∧ (w < 3)` era idénticamente falso. Medido: 0 aristas ocultas en la vista inicial. El mismo bug anulaba la escala de grosores (todas las aristas salían con el grosor máximo) y el resaltado por grosor.
5. **Caché de servidor dependiente de la plataforma.** El endpoint `/api/mapa-legal-graph` cacheaba el grafo con `unstable_cache` del framework, cuyo almacén *descarta silenciosamente toda entrada mayor a 2 MB*. Con el corpus de producción (1,34 MB) funcionaba; al crecer el corpus local a 2,85 MB serializados, cada solicitud recomputaba el grafo y el layout completos: 75–96 s por request, contra un timeout de 60 s en el cliente. Resultado: el mapa directamente no cargaba.

3.1 Diagnóstico experimental: el costo es pintar, no el DOM

La hipótesis inicial ("hay demasiados elementos en el DOM; hay que virtualizar") se refutó con un experimento de ablación medido con Chrome headless (14-ago-2026, build de producción):

Tabla 2. Ablación del render a n = 3.702, m = 7.006. La última fila es la observación decisiva.

ESCENARIO	ELEMENTOS SVG EN EL DOM	FPS
Sistema completo	18.112	3
Sin las 3.702 etiquetas	14.412	4,5
Sin etiquetas ni las 7.006 aristas	7.406	32
Solo 300 vértices	602	61
Los mismos 18.112 elementos con `display:none`	**18.112**	**61**

Con el árbol DOM íntegro pero sin rasterizar, el sistema alcanza 61 FPS: **el cuello de botella no es la estructura, es la pintura**. Y dentro de la pintura, las aristas dominan: quitarlas multiplica el rendimiento por 7 (4,5 → 32 FPS), mientras que quitar las etiquetas apenas lo mueve. Para cuantificarlo se midió la

frecuencia de cuadro variando solo la cantidad de aristas SVG dibujadas, con los vértices fijos en 3.702 (Figura 1). El tiempo de cuadro $T(E)$ = 1000/FPS ajusta un modelo lineal:

$$T_{\text{cuadro}}(E) \approx 24{,}3\ \text{ms} + 0{,}0598\ \text{ms} \cdot E \qquad (R^2 = 0{,}973) \tag{1}$$

Es decir: **cada arista SVG cuesta ≈59,8 µs de rasterización por cuadro**, unas 30 veces más que un círculo (quitar las 2.261 aristas del overview llevaba de 8 a 46–54 FPS; quitar los 3.702 círculos, de 54 a 61). El presupuesto para sostener 30 FPS (33,3 ms por cuadro) se deduce de (1): $E \le (33{,}3 - 24{,}3)/0{,}0598 \approx$ **150–230 aristas SVG**. Ningún nivel de detalle razonable de un grafo de 9.000 aristas cabe en ese presupuesto: la conclusión no es "ocultar más", es *abandonar SVG para las aristas* (§5.4).

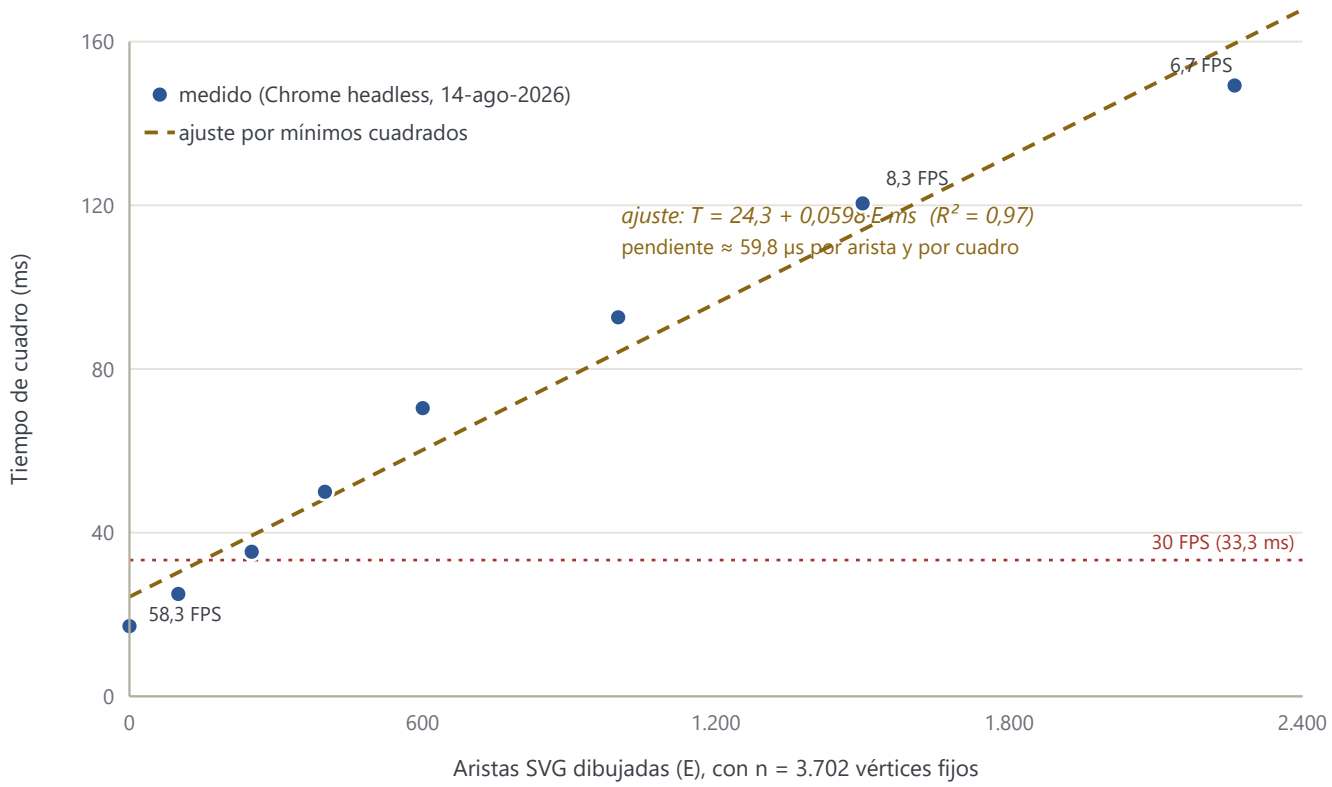


Figura 1. Tiempo de cuadro medido en función de la cantidad de aristas SVG dibujadas (vértices fijos en 3.702). El modelo lineal (ec. 1) explica el 97,3 % de la varianza y da un costo marginal de ≈59,8 µs por arista y por cuadro. La línea de 30 FPS muestra por qué ningún LOD alcanza mientras las aristas sean elementos SVG: el presupuesto es de ~150–230 aristas.

4 Metodología

Todas las mediciones se hicieron con Chrome headless sobre el build de producción del frontend (no el modo de desarrollo, cuyo overhead distorsiona), con scripts reproducibles (`scratchpad/medir-mapa*.mjs`, `medir-layout.mjs`, `verificar-fijo.mjs`). Las cifras 3D usan el rasterizador por software SwiftShader (peor caso: sin GPU). Las latencias de servidor se midieron con `curl` contra el stack local en Docker (frontend :4000, backend :4600, PostgreSQL 17), corpus de 8.034 normas. Cada optimización se midió con A/B limpio:

mismo método, mismos flags, alternando solo el componente bajo estudio (vía `git stash`). Dos trampas de medición documentadas: (a) `document.querySelector("svg")` devuelve un ícono del encabezado, no el grafo — hay que seleccionar el SVG que contiene `<circle>`, o se miden 61 FPS falsos; (b) tras soltar el mouse en 3D la imagen sigue moviéndose por la inercia de los controles de cámara, no por la física — confundirlas invalida el A/B.

5 Las transformaciones

5.1 Ponderación semántica de aristas: reparar el LOD desde los datos

La primera intervención no fue de render sino de *modelo*: reemplazar el peso constante por una función w: $T \to \{1,\ldots,10\}$ que codifica la fuerza jurídica del vínculo — DEROGA y DEROGA_PARCIALMENTE: 10; VETA: 9; MODIFICA: 8; REGLAMENTA e IMPLEMENTA: 7; RATIFICA: 6; COMPLEMENTA, SUPLETORIA y ESPECIALIZA: 5; RELACIONADA: 2; referencia inline: 1. Esto revivió el predicado de arista débil (las 4.745 RELACIONADA del corpus del diagnóstico, el 67,7 % de m, pasaron a ser ocultables: 7.006 → 2.261 aristas en la vista general) y restauró la escala de grosores (de 1 valor a 7). La lección metodológica: *un LOD puede estar muerto sin producir ningún error*; el síntoma era únicamente `hiddenLines: 0` en una métrica interna. La ponderación además alimenta la simulación 3D y el minimapa de guías, de modo que la semántica jurídica quedó codificada una sola vez, en el servidor.

5.2 Nivel de detalle con umbral invariante de escala

El LOD original comparaba la escala de zoom k contra un umbral absoluto $u = 0{,}45$: por debajo, las aristas débiles se ocultan. El defecto es que la escala del encuadre inicial (*fit-to-view*) depende del viewport:

$$k_{\text{fit}}(W, H) = 0{,}85 \cdot \min\left(W / (1800 + 160),\ H / (1200 + 160) \right) \tag{2}$$

donde 1800 × 1200 es la caja canónica del layout (§5.5) y 160 el margen. En un lienzo de 1400 × 700, $k_{\text{fit}} = 0{,}438$ — a dos centésimas del umbral; en uno de 1200 × 600, $k_{\text{fit}} = 0{,}375$, claramente debajo. La visibilidad del 65 % del grafo (las 5.929 aristas RELACIONADA) dependía entonces del *signo de* $0{,}45 - k_{\text{fit}}(W,H)$: una función del tamaño de la ventana del usuario y del tamaño del corpus (que agranda la caja y deprime k_{fit}), no de ninguna decisión de diseño. El umbral se había calibrado con 3.700 nodos y quedó inalcanzable al crecer el corpus.

La corrección hace el umbral *relativo al encuadre*: $u = \beta \cdot k_{\text{fit}}$ con $\beta = 1{,}3$ (y piso absoluto 0,45 como salvaguarda). La condición de visibilidad total de aristas pasa a ser

$$k \geq \beta \cdot k_{\text{fit}} \iff k / k_{\text{fit}} \geq \beta \tag{3}$$

es decir, depende solo del *factor de acercamiento relativo* del usuario, que es adimensional e invariante ante el tamaño del viewport y del corpus. Es el mismo principio por el que los mapas cartográficos definen sus niveles por razón de escala y no por píxeles. En la vista general no se dibujan cero aristas débiles: se dibuja un **presupuesto fijo** de 1.400, priorizando las incidentes a hubs (las que "dan forma" al grafo). Esto acota el trabajo de pintura del overview por

$$E_{\text{dibujadas}} \le E_{\text{fuertes}} + 1.400 = (m - m_{\text{RELACIONADA}}) + 1.400 \tag{4}$$

que en el corpus actual da ≤ 4.608 y, crucialmente, *crece sublinealmente con m*: el término dominante del crecimiento del corpus son precisamente las RELACIONADA débiles, que quedan fuera del presupuesto.

5.3 Culling por viewport: Liang–Barsky con pre-test de caja

Al hacer zoom, solo una fracción del grafo es visible. El recorte ingenuo por caja envolvente (¿se solapa el rectángulo de la arista con el viewport?) falla en este grafo por la heterogeneidad de grado: los hubs emiten aristas que atraviesan el mapa entero, y la caja envolvente de una diagonal larga se solapa con casi cualquier viewport aunque el segmento pase lejos. Medido: con 10 vértices en pantalla quedaban 643 aristas "visibles" por caja. La solución es el test exacto de Liang–Barsky [3], que parametriza el segmento como $P(t) = A + t(B - A)$, $t \in [0,1]$, y recorta el intervalo contra las cuatro semiplanos del rectángulo en O(1) con 4 divisiones como máximo. El pipeline por arista es: (i) pre-test de caja envolvente (4 comparaciones, rechaza la mayoría), (ii) Liang–Barsky solo para las sobrevivientes. Con zoom, quedan **51 aristas dibujadas de 7.006** (0,7 %).

Tres detalles de implementación resultaron indispensables, y los tres son lecciones de análisis amortizado:

- **Early-exit global:** si el viewport contiene la caja del grafo completo (el caso de la vista inicial, que es además el más frecuente), el culling se saltea por completo. Sin esto, el culling costaba *más* que el repintado que ahorraba, precisamente donde no ahorraba nada.
- **Escritura incremental:** reasignar `display` a los 10.706 elementos en cada cuadro cuesta 5–15 ms aunque el valor no cambie (el estilo invalida). Se mantiene un `Uint8Array` con el estado aplicado y solo se escribe la diferencia.
- **Orden de los tests:** caja primero (barata, alta selectividad), test exacto después. El costo esperado por arista queda dominado por las 4 comparaciones del pre-test.

Resultado aislado de esta etapa: 3 → 8 FPS (×2,7). Necesario pero insuficiente — consistente con la ecuación (1): mientras la arista sobreviviente sea un elemento SVG, su costo unitario de ~60 µs no cambia.

5.4 Render híbrido: aristas en canvas, vértices en SVG

La ecuación (1) admite una sola salida: cambiar el término $c_e \approx 59{,}8$ µs, no el factor E. La diferencia de fondo entre SVG y canvas es de modelo de ejecución: SVG es *modo retenido* (cada arista es un objeto vivo del DOM con estilo, eventos e invalidación propios; el navegador re-rasteriza primitivas individuales en cada transformación), canvas es *modo inmediato* (un solo bitmap; el costo es proporcional a los píxeles pintados, no a los objetos). La arquitectura final es híbrida:

- Un `<canvas>` *detrás* del SVG (`pointerEvents: none`) dibuja todas las aristas, compartiendo la transformación de cámara vía `ctx.setTransform`.
- Los vértices y etiquetas permanecen en SVG: el *hit-testing* (click, hover, accesibilidad) lo resuelve el navegador gratis. Moverlos a canvas exigiría picking manual con quadtree — se evaluó y se difirió (§9).
- **Lotes por grupo de estilo:** cambiar `strokeStyle` o `globalAlpha` cierra el lote interno del rasterizador. Las aristas visibles se agrupan por la clave (color, opacidad, grosor) y se emite *un solo* `stroke()` por grupo. Las llamadas de dibujo pasan de $O(E_v)$ a $O(S)$, con $S \le 7$ grupos en la práctica (uno por tipo de vínculo visible): mil segmentos en un `stroke()` cuestan una fracción de mil `stroke()` sueltos.
- El culling de §5.3 vive *dentro* de la pasada de dibujo — es la misma iteración, así que sale gratis. Las flechas de derogación (antes un `<marker>` SVG) se dibujan a mano solo con $k > 0{,}3$.

Resultado medido: 8 → 32,7 FPS (build de producción), con los elementos SVG reducidos de 18.112 a 11.102. Componiendo las etapas: 3 → 32,7 FPS, una mejora de ×10,9 a igual corpus.

5.5 Precómputo del layout en el servidor

La simulación de fuerzas de d3-force integra por Verlet de velocidad [5] con tres fuerzas de costo no trivial por tick: la repulsión de muchos cuerpos y la colisión usan el algoritmo de Barnes–Hut [1] sobre un quadtree — $O(n \log n)$ por tick con parámetro de apertura θ — y la fuerza de resorte es $O(m)$. El costo total del layout es entonces

$$C_{\text{layout}} = k \cdot O(n \log n + m), \quad \text{con } k = 240 \text{ ticks} \tag{5}$$

Medido en el servidor: 16 s a 66,5 ms/tick con n = 3.702 (y ~80–88 s con n = 8.034). La transformación no reduce (5): **cambia quién y cuándo lo paga**. El layout corre una vez por corpus en una caja canónica fija de 1800 × 1200 (con la Constitución Nacional anclada al centro y anclas de categoría sobre un anillo de radio $0{,}32 \cdot \min(W,H)$), y el payload viaja con las posiciones (x, y) redondeadas a un decimal. El cliente, si detecta posiciones precomputadas, *no crea la simulación* (0 ticks en vez de 240) y solo aplica un encuadre afín — $O(n)$. El costo por visita pasa de la ecuación (5) a $O(n)$, y el costo (5) se amortiza entre todas las visitas del período de validez del corpus.

Dos consecuencias de producto, no solo de performance: (a) **determinismo** — todos los usuarios ven el mismo mapa, y un vértice pasa a ser un lugar estable ("la 24.013 está en el barrio laboral, arriba a la izquierda") en vez de un resultado aleatorio por visita; (b) se eliminó el *drag* de vértices: con el layout como información (cercanía = afinidad temática), permitir moverlos solo la destruía. De yapa midió ~8 → ~10 FPS y el click bajó de 1,5 a 0,4 ms al no recalentar la física.

5.6 Interacción en O(grado): el índice de adyacencia

El resaltado ("spotlight") de un vértice recorría las m aristas con `each()` para armar el conjunto de vecinos: $O(m)$ por hover o click. Con un índice de adyacencia precalculado (mapa id → lista de vecinos), la operación es $O(\deg(v))$ — 2 en la mediana, 883 en el peor caso (los hubs) —, más el costo de aplicar estilos al

subconjunto. Click medido: 1,5 → 0,4 ms. Es la desigualdad básica $\deg(v) \ll m$ en grafos dispersos (aquí $m/n \approx 1{,}14$) explotada en el lugar correcto.

5.7 La variante 3D: presupuestos y siembra determinística

El mapa 3D estaba peor que el 2D: **50.123 ms de bloqueo del hilo principal al montar**. El costo dominante era un `SpriteText` por vértice — 3.700 lienzos de texto, cada uno con su textura subida a la GPU —, seguido por las 7.006 aristas dibujadas con flechas, 90 ticks de warmup de física 3D y el antialiasing. Las intervenciones siguen el mismo patrón de presupuesto: etiquetas solo para los ~120 vértices más importantes (más los vecinos del seleccionado), aristas *cargadas* todas (la física las necesita) pero *dibujadas* solo con $w \geq 3$ vía `linkVisibility`, flechas solo en DEROGA, `antialias: false` y esferas de 6 segmentos. El warmup bajó de 90 a 25 ticks porque las posiciones **se siembran desde el layout 2D canónico**: (x, y) escaladas y una coordenada z determinística por hash del id — no `Math.random()`, que cambiaría el mapa por visita y rompería el determinismo de §5.5. Al asentarse, las posiciones se congelan (`fx/fy/fz`) y filtrar por categoría no reacomoda nada. Resultados: bloqueo 50.123 → ≈230 ms (×218), FPS en reposo 4,7 → 20–33 (SwiftShader, sin GPU), memoria JS 90 → 56–66 MB.

5.8 Servido: caché con coalescencia de solicitudes y compresión en el codo

El modo de falla del caché de plataforma (§3, punto 5) es instructivo: un límite silencioso (entradas > 2 MB se descartan sin error) convirtió un endpoint O(1) en uno que ejecutaba la ecuación (5) *por solicitud*. La corrección reemplaza el caché de plataforma por un caché en memoria del proceso con TTL de 10 días que almacena **la promesa** del resultado, no el resultado:

$$\text{solicitudes concurrentes durante el cómputo: coste}(\lambda, \tau) = 1 \text{ cómputo, no } \lambda \cdot \tau \tag{6}$$

Si λ solicitudes llegan durante la ventana de cómputo τ (aquí $\tau \approx 80$ s), todas esperan la *misma* promesa: la coalescencia elimina la estampida (*thundering herd*) que con el caché roto multiplicaba el trabajo. Si el cómputo falla, la promesa se descarta y la siguiente solicitud reintenta. El costo por solicitud en régimen es O(1): servir un buffer ya serializado.

La compresión se aplica a mano (el framework no comprime los route handlers) y el nivel se eligió midiendo la curva calidad–CPU sobre el payload real: gzip-6 = 229 kB en 21 ms; **Brotli q5 = 195 kB en 27 ms**; Brotli q11 = 167 kB en 2.297 ms. El codo está en q5: q11 compra 28 kB al precio de 2,3 s de CPU — 85× más tiempo por un 14 % menos de bytes. Los buffers comprimidos (br y gzip) se memoizan junto al JSON y se sirven según `Accept-Encoding`, con `Vary` para no envenenar cachés intermedios. Sobre el corpus actual: 2.515.943 → 371.817 bytes (×6,8). Completan el esquema un `s-maxage` de 10 días en el CDN (que se purga en cada despliegue) y un caché de 6 horas en `localStorage` del cliente: **tres capas**, cada una con su vigencia y su mecanismo de invalidación explícito.

6 Resultados consolidados

Tabla 3. Resultados extremo a extremo. Entornos: (a) Chrome headless, build de producción, n = 3.702; (b) ídem con SwiftShader; (c) stack local Docker, n = 8.034, corpus completo.

MÉTRICA	ANTES	DESPUÉS	MEJORA	ENTORNO
FPS en interacción, mapa 2D	3	32,7	×10,9	(a)
Aristas dibujadas en la vista general	7.006 (LOD muerto)	2.261 + presupuesto	−68 %	(a)
Aristas dibujadas con zoom (10 vértices en pantalla)	643 (solo caja)	51 (Liang–Barsky)	−92 %	(a)
Latencia de click/spotlight	1,5 ms	0,4 ms	×3,8	(a)
Ticks de física pagados por el cliente	240	0	–	(a)
Bloqueo del hilo principal al montar, 3D	50.123 ms	≈230 ms	×218	(b)
FPS en reposo, 3D (sin GPU)	4,7	20–33	×4–7	(b)
Memoria JS, 3D	90 MB	56–66 MB	−27–38 %	(b)
Latencia del endpoint, en régimen	75–96 s *(caché roto)*	0,35 s	**×215–275**	(c)
Transferencia del grafo	2.516 kB	372 kB (Brotli q5)	×6,8	(c)
Estabilidad del layout (drag + filtro + zoom)	se reacomodaba	0 de 3.702 vértices movidos	determinístico	(a)

7 Análisis asintótico consolidado

La tabla 4 resume el argumento central: cada operación del sistema cambió de clase, no de constante. Notación: $n = |V|$, $m = |E|$, $k = 240$ ticks, E_v = aristas visibles tras culling y LOD, S = grupos de estilo (≤ 7), $\deg(v)$ = grado del vértice, λ = solicitudes concurrentes.

Tabla 4. Complejidad por operación, antes y después. Las filas marcan quién paga el costo.

OPERACIÓN	ANTES	DESPUÉS	QUIÉN LO PAGA AHORA
Layout de fuerzas	$O(k\cdot(n \log n + m))$ *por visita, en el cliente*	$O(k\cdot(n \log n + m))$ *una vez por corpus*; cliente $O(n)$	el servidor, amortizado
Respuesta del endpoint	$O(k\cdot(n \log n + m))$ por solicitud (caché >2 MB descartado)	$O(1)$ amortizado; λ concurrentes = 1 cómputo (ec. 6)	buffer precomprimido
Pintura de aristas por cuadro	$c_{SVG}\cdot m$, $c_{SVG} \approx 59{,}8$ µs (ec. 1)	$c_{canvas}\cdot E_v + O(S)$ draw calls, $E_v \le E_{fuertes} +$ 1.400 (ec. 4)	acotado por presupuesto
Culling por arista	– (no había)	$O(1)$: 4 comparaciones + Liang–Barsky condicional	dentro de la pasada de dibujo
Hover / click (spotlight)	$O(m)$	$O(\deg(v))$ – mediana 2	índice de adyacencia
Cambio de filtro	reconstrucción del DOM $O(n + m)$	toggle con escritura incremental (solo diferencias)	Uint8Array de estado
Etiquetas (2D / 3D)	n sprites/textos	≤ 160 / ≤ 120 + vecinos del seleccionado	presupuesto por centralidad
Transferencia	JSON crudo $O(n + m)$	$O(n + m)/6{,}8$ + caché CDN 10 d + localStorage 6 h	Brotli q5, tres capas

La consecuencia práctica se enuncia como propiedad de escalabilidad: **el costo por cuadro está acotado por una constante que no depende de m** (ec. 4: el presupuesto de débiles es fijo y las aristas fuertes crecen mucho más despacio que el corpus, porque el crecimiento marginal del corpus es dominado por RELACIONADA de peso 2), y **el costo por solicitud está acotado por una constante que no depende ni de n ni de m** (buffer precomprimido). El único costo que sigue creciendo con el corpus — la ecuación (5) — se paga una vez por corpus, en el servidor, fuera de toda ruta interactiva. Así, el corpus multiplicó su tamaño por 5,3 mientras la latencia percibida del sistema *bajó* dos órdenes de magnitud.

8 Limitaciones

1. **Los 32,7 FPS se midieron con n = 3.702.** El corpus local ya tiene 8.034 vértices y 9.137 aristas; el modelo de costo predice que el régimen se sostiene (las aristas nuevas son mayormente débiles y quedan fuera del presupuesto de la ec. 4, y los círculos SVG cuestan ~30× menos que las aristas que ya no se pintan en SVG), pero la verificación formal en navegador a esa escala está pendiente.
2. **Arranque en frío.** La primera solicitud tras un reinicio o despliegue paga la ecuación (5) completa (~88 s con n = 8.034), más que el timeout de 60 s del cliente: el primer visitante fallaría. Mitigación

operativa: calentar el caché con una solicitud (`curl`) tras cada despliegue. Mitigación estructural pendiente: computar el layout en el momento de la carga del corpus, no de la primera solicitud.

3. **El backend aún recomputa el grafo base (~1 s) por solicitud de `/laws/graph`**; está amortiguado por el caché del frontend pero la memoización en el propio backend está pendiente.
4. **Una sola máquina de medición.** Las cifras absolutas (FPS, ms) son de un equipo de desarrollo con rasterización por software en el caso 3D; las *razones* de mejora son el resultado robusto, las cifras absolutas variarán por hardware.
5. **El determinismo tiene un costo de frescura:** el TTL de 10 días implica que una norma cargada hoy puede tardar hasta 10 días en aparecer en el mapa si no media un despliegue o reinicio. Es una decisión deliberada (el grafo cambia lentamente) y reversible por configuración.

9 Trabajo futuro

- **Vértices en canvas con picking por quadtree** (o render WebGL completo): el modelo de costo indica que es la vía hacia 60 FPS sostenidos; exige reimplementar hit-testing y accesibilidad que hoy el SVG regala.
- **Web Worker para el layout residual** (Kelsen y minimapas aún simulan en el cliente) y para el parse del payload.
- **LOD por centralidad global** (PageRank sobre G) en lugar de grado local, para que el overview muestre las normas estructuralmente importantes y no solo las más conectadas.
- **Clustering jerárquico** (agregar barrios en super-vértices al alejar), que convertiría la cota de la ec. (4) en logarítmica respecto del corpus.
- **Precómputo del layout en el pipeline de carga del corpus**, eliminando el arranque en frío (§8.2).

10 Conclusión

Un grafo normativo nacional es un objeto que crece sin pedir permiso: cada ley nueva agrega un vértice y, en promedio, poco más de una arista — casi siempre débil. La versión original del Mapa Legal pagaba ese crecimiento en el peor lugar posible: en cada cuadro de cada interacción de cada visitante. El trabajo descripto movió cada costo a su lugar correcto — el layout al servidor y al momento del corpus; la pintura a un presupuesto fijo gobernado por semántica jurídica; la interacción al grado del vértice; el servido a un buffer inmutable — y el resultado es un sistema cuya complejidad percibida es constante aunque el objeto subyacente se quintuplicó. Ninguna técnica empleada es nueva en sí misma (Barnes–Hut es de 1986, Liang–Barsky de 1984, el modo inmediato es tan viejo como los framebuffers); lo que este caso aporta es la evidencia medida de su composición sobre un dato público real, y el registro de los modos de falla — el LOD muerto sin error visible, el caché que descarta en silencio, el umbral absoluto que dependía del tamaño de la ventana — que son, quizás, lo más transferible a otros equipos que construyan infraestructura cívica con presupuesto de una sola persona.

11 Referencias

1. Barnes, J. y Hut, P. (1986). "A hierarchical O(N log N) force-calculation algorithm". *Nature*, 324, 446–449.
2. Fruchterman, T. M. J. y Reingold, E. M. (1991). "Graph Drawing by Force-directed Placement". *Software: Practice and Experience*, 21(11), 1129–1164.
3. Liang, Y.-D. y Barsky, B. A. (1984). "A New Concept and Method for Line Clipping". *ACM Transactions on Graphics*, 3(1), 1–22.
4. Bostock, M., Ogievetsky, V. y Heer, J. (2011). "D³: Data-Driven Documents". *IEEE Transactions on Visualization and Computer Graphics*, 17(12), 2301–2309.
5. Verlet, L. (1967). "Computer 'Experiments' on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules". *Physical Review*, 159(1), 98–103.
6. Alakuijala, J. y Szabadka, Z. (2016). "Brotli Compressed Data Format". *RFC 7932*, IETF.
7. Barabási, A.-L. y Albert, R. (1999). "Emergence of Scaling in Random Networks". *Science*, 286(5439), 509–512.
8. Kelsen, H. (1934). *Reine Rechtslehre* [Teoría pura del derecho]. Deuticke, Viena. (Fundamento del layout jerárquico "pirámide de Kelsen" del sistema.)

Reproducibilidad: los scripts de medición (`scratchpad/medir-mapa*.mjs`, `medir-layout.mjs`, `verificar-fijo.mjs`) y el código del sistema (`LegalGraph2D.tsx`, `LegalGraph3D.tsx`, `legalGraphLayout.ts`, `app/api/mapa-legal-graph/route.ts`, `laws.service.ts`) acompañan al repositorio del proyecto. Mediciones de agosto de 2026: cliente con Chrome headless sobre build de producción (3D con SwiftShader); servidor sobre stack Docker local (frontend :4000, backend :4600, PostgreSQL 17) con corpus de 8.034 normas. Contacto: Observatorio de Leyes.