Manual de Documentación Técnica de Código

Proyecto:
Ciencias de la Computación II
Autores:
Tomás Alejandro Delgado Ortíz - 20221020045
Juan Pablo Mosquera Marín - 20221020026

Este documento compila la documentación técnica detallada de los módulos del proyecto explicando su fucionalidad y principales algoritmos implementados con la finalidad de brindar soporte al trabajo de construcción del codigo y facilitar la escalabilidad del mismo.

Tabla de contenidos

BinariaExterna.vue

Archivos Creados

1. BinariaExterna.vue

Ubicación: `/frontend/src/views/external-searches/BinariaExterna.vue`

Qué hace (resumen):

- Implementa la búsqueda binaria externa en bloques (archivo por bloques) con inserción ordenada y corrimientos.
- Permite configurar `capacidad` y `digitosClave`, crear la estructura (bloques de tamaño ⌊√capacidad⌋) y realizar `insertar`, `buscar`, `eliminar`, `importar` y `exportar`.

- Visualiza bloques verticales con índices globales y resalta posicionamientos de búsqueda.

Algoritmos y funcionalidades principales

Creación y visualización de la estructura

1. **Crear estructura**: valida `capacidad >= 10` y `digitosClave >= 1`, calcula `elementosPorBloque = ⌊√capacidad⌋` y `numeroBloques = ceil(capacidad / elementosPorBloque)`.
2. **Estructura interna**: `estructura` es un array de bloques `[(number|null)[]]` donde se mantienen elementos ordenados por bloque.
3. **Visualización**: muestra bloques y usa `getDisplayIndicesForBlock` para limitar la renderización en bloques grandes.

ResiduoHuffman.vue

Archivos relacionados

1. `frontend/src/views/ResiduoHuffman.vue`

Ubicación: `/frontend/src/views/ResiduoHuffman.vue`

Qué hace (resumen):

- Construye y visualiza un árbol de Huffman a partir de un `inputText` proporcionado por el usuario.
- Calcula frecuencias de caracteres, genera códigos prefijo (mapa `codes`), cifra y descifra textos, y permite exportar/importar la configuración y el árbol.
- Muestra métricas pedagógicas (tabla de frecuencias, longitud ponderada L, tasa de compresión aproximada) y provee una vista gráfica interactiva del árbol con `vis-network`.

Algoritmos y funcionalidades principales

1) Cálculo de frecuencias

- Lee la cadena `inputText` y genera un `Map<string, number>` con la frecuencia de cada símbolo.
- Normaliza la entrada (opcionalmente elimina saltos de línea si el usuario marca esa opción) antes del conteo.

2) Construcción del árbol de Huffman

- Crea nodos tipo `{ char?: string, freq: number, left?: Node, right?: Node }` y usa una cola de prioridad (implementada con arreglo y sort o con pequeño heap local) para combinar los dos nodos de menor frecuencia hasta obtener un único árbol raíz.
- Devuelve `HuffmanResult` con `tree`, `frequencies` (array ordenada) y `codes` (mapa `char -> binario`).

3) Generación de códigos y codificación

- Recorre el árbol recursivamente y asigna `0`/`1` a las ramas (convención: izquierda `0`, derecha `1`).
- `codes` es un objeto/Map usado para: `encodeText(text, codes)` → devuelve cadena binaria (string) y `decodeBits(bits, tree)` → reconstruye el texto.

4) Métricas y comprobaciones

- Calcula `L = Σ(frecuencia(char) * longitud(codigo(char)))` y la muestra junto con el tamaño en bits del texto codificado.
- Opcionalmente muestra ratio de compresión comparando `inputText.length * 8` vs `encodedBits.length`.

5) Visualización

- Usa `vis-network` para renderizar nodos y aristas; las hojas muestran el carácter y la frecuencia, las aristas se etiquetan con `0` o `1`.
- Al seleccionar un nodo se resalta el camino desde la raíz y se puede inspeccionar el subárbol.

6) Import / Export

- Exporta un JSON con `type: 'huffman'`, `frequencies`, `tree` (representación serializable de nodos) y `config` (por ejemplo `originalText` o parámetros de normalización).
- Importa validando `type === 'huffman'` y reconstruye `huffmanResult`, `inputText` y la vista gráfica.

Parámetros de configuración y validaciones

- `inputText`: string no vacío requerido para construir el árbol.
- Validaciones:
  - No construir con texto vacío.
  - Validar formato del JSON importado y `type`.
  - Manejar caracteres no ASCII (se guarda exactamente la cadena y la frecuncia por símbolo unicode).

API interna (funciones / utilidades observadas)

- `calculateFrequencies(text): Map<string,number>`
- `buildHuffmanTree(frequencies): Node` (cola de prioridad)
- `generateCodes(tree): Record<string,string>`
- `encodeText(text, codes): string`
- `decodeBits(bits, tree): string`
- `createExportData('huffman', config, data)` y `downloadJsonFile(...)` (utilidades comunes de import/export)

Uso y comportamiento UX

- Pasos típicos: pegar texto → `Construir` → revisar `tabla de frecuencias` y `L` → `Codificar / Decodificar` → `Exportar` si se desea.
- Los botones de `Exportar`/`Decodificar` están deshabilitados hasta que exista un `huffmanResult` válido.

Consideraciones de rendimiento

- La construcción es O(n log n) respecto a la cantidad de símbolos distintos (n = número de símbolos únicos en la entrada).

- Codificación/decodificación es O(T) donde T es la longitud del texto.

ResiduoMultiple.vue

Archivos relacionados

1. `frontend/src/views/ResiduoMultiple.vue`
2. `frontend/src/utils/residueMultiTree.ts` (implementación de nodos y utilidades: `RMNode`, `insertRM`, `searchRM`, `buildRMTemplate`, `groupBits`)

Ubicación: `/frontend/src/views/ResiduoMultiple.vue`

Qué hace (resumen):

- Implementa una estructura tipo árbol de residuos múltiples que agrupa claves (letras A-Z en la interfaz) según una codificación binaria dividida en grupos de `m` bits (parámetro `m` configurable entre 1 y 3).
- Soporta crear la plantilla del árbol (`buildRMTemplate(m)`), insertar claves (`insertRM`), buscar (`searchRM`), borrar (limpia la hoja) y exportar/importar la estructura completa.
- Visualiza la estructura en modo gráfico (`vis-network`) o en modo textual, muestra el código binario y la partición en grupos para la clave actual.

Algoritmos y funcionalidades principales

1) Plantilla y estructura de nodos

- `RMNode` representa un nodo con campos: `key`, `code`, `labelBits`, `children[]`.
- `buildRMTemplate(m)` costruye un árbol multinivel donde cada conector tiene `2^m` hijos (por ejemplo, m=2 → 4 hijos por nodo) hasta una profundidad suficiente para cubrir las codificaciones de letras (se maneja para 26 letras con `digitosClave=1` pero el template es general).

2) Conversión letra → código binario → grupos

- `letterToCode(letter)` (desde `digitalTree.ts`) convierte la letra en un código binario fijo.
- `groupBits(code, m)` divide ese código en grupos binarios de longitud `m` (p. ej. `101100` → [`10`, `11`, `00`] para m=2).

3) Inserción (`insertRM`)

- Se recorre el árbol por los grupos en orden: para cada grupo se toma el índice `parseInt(group,2)` y se avanza al `children[idx]`.
- Si se llega a un nodo hoja vacío se asigna `key` y `code`.
- La función devuelve `{ root, path, status }` donde `path` es la secuencia de índices recorrida y `status` indica `inserted` o `duplicate`.

4) Búsqueda (`searchRM`)

- Mismo recorrido por `groupBits` hasta encontrar la hoja con `key === letter` o concluir no encontrada; devuelve `{ found, path }`.

5) Eliminación

- La vista implementa eliminación lógica (limpia `key` y `code` en la hoja encontrada) y actualiza mensajes/selección.

6) Render gráfico

- `buildGraphData(node)` recorre `RMNode` y construye arrays `nodes` y `edges` para `vis-network`.
- Distinción visual: nodos con `key === null` son conectores (box) con `labelBits`; hojas con `key` son círculos con la letra.
- `renderGraph()` crea/actualiza el `Network` y centra en el nodo objetivo (`computeTargetId()` usa `pathIdx`).

7) Import / Export

- Exporta con `createExportData('residuo-multiple', config, root)` donde `config` incluye `base: m` y `digitosClave`.
- Importa validando mediante `validateResidueMultipleImport(importData)` y restaura `m`, `root` y el estado `treeCreated`.

Parámetros y validaciones

- `m` (1..3): bits por grupo; `treeCreated` controla disponibilidad de operaciones.
- Validaciones principales:
  - `input` debe ser una letra A-Z (la vista fuerza `toUpperCase()` y limita longitud a 1).
  - Al importar valida estructura con `validateResidueMultipleImport` para evitar JSON malformados.

UX y mensajes

- Interfaz muestra: panel de configuración (selección de `m`), contenedores para código binario y `groups`, controles `Insertar/Buscar/Borrar`, y botones `Guardar/Abrir`.
- Mensajes informativos describen la operación (ruta recorrida, grupos, duplicado, insertado, borrado).

Consideraciones de implementación

- El árbol es estático en profundidad tras `buildRMTemplate`, lo que facilita render determinista y posicionamiento en `vis-network`.
- La eliminación es lógica (vacía la clave); no se realiza recolocación de claves.

ResiduoSimple.vue

Archivos relacionados

1. `frontend/src/views/ResiduoSimple.vue`

Ubicación: `/frontend/src/views/ResiduoSimple.vue`

Qué hace (resumen):

- Implementa la variante mínima de tabla por residuos: tabla indexada por `h(k) = k mod m` para claves numéricas o transformaciones equivalentes para letras (p. ej. `A→0...`).
- Permite seleccionar estrategia de resolución (encadenamiento o sondeo), observar métricas de carga (load factor), insertar, buscar, eliminar, y exportar/importar la tabla.

Algoritmos y funcionalidades principales

1) Función hash y mapeo de claves

- `h(k) = k mod m` o, para letras, `code = letterToNumeric(letter); h = code % m`.

2) Resolución de colisiones

- Encadenamiento: cada entrada `table[i]` es una lista/array de elementos; la inserción añade al final si no existe duplicado.
- Sondeo lineal (opcional): en caso de colisión se busca la siguiente posición libre `i = (i+1) % m` hasta completar o encontrar duplicado.

3) Import / Export

- Exporta JSON con `type: 'residuo-simple'`, `config` y `data` (tabla/encadenamientos).
- Importa validando `type` y la forma del objeto antes de restaurar en el estado local.

Parámetros y validaciones

- `m` (tamaño de la tabla), `strategy` (`encadenamiento` | `sondeo`), y `maxLoadFactor` opcional para alertas.
- Validaciones: claves válidas, `m >= 2`, y en `sondeo` se controla la condición de búsqueda completa para evitar ciclos infinitos.

Notas finales

- Es un componente de referencia para comparar comportamiento frente a `ResiduoMultiple` y otras técnicas; simple y con código claro para didáctica.

App.vue

Archivos relacionados

1. `frontend/src/App.vue`

Ubicación: `/frontend/src/App.vue`

Qué hace (resumen):

- Componente raíz de la SPA; monta `NavBar`, el `router-view` y contiene capas de estilo/globales y manejo básico de import/export a nivel de aplicación.

Responsabilidades y detalles de implementación

Layout y navegación

- Renderiza el `NavBar` en la parte superior y el contenido de cada ruta mediante `<router-view/>`.
- Contiene la lógica mínima para toggles globales (tema oscuro/claro) y escucha de eventos de ruta para labores de analítica o limpieza.

Integraciones globales

- Importa estilos globales (ej. `vis-network` CSS) y registra proveedores o plugins aplicables (por ejemplo, i18n o store si existiera).
- Provee un punto central para funciones cross-cutting: abrir diálogos modales globales, hooks de import/export compartidos y manejo de notificaciones.

Buenas prácticas y notas

- Mantener `App.vue` lo más pequeño posible: delegar la lógica a composables (`useNavigation`, `useImportExport`) y a componentes de presentación.
- Cambios en `App.vue` afectan la navegación y la carga de librerías globales, por lo que las modificaciones deben revisarse con pruebas de navegación.

---

2. **Floyd-Warshall**: triple bucle (k,i,j) actualiza `matriz[i][j] = min(matriz[i][j], matriz[i][k]+matriz[k][j])`.
3. **Cálculos derivados**: excentricidades (máxima distancia finita por vértice), mediana (mínima excentricidad), centros (vértices con excentricidad mínima) y diámetro (excentricidad máxima).

Notas finales

- Interfaz rica y orientada a educación; incluye validaciones de entrada y formato de aristas, y renderiza la matriz de distancias con notación ∞ para caminos no alcanzables.

HashCambioBaseExterno.vue

Archivos Creados

1. HashCambioBaseExterno.vue

Ubicación: `/frontend/src/views/external-searches/hash/HashCambioBaseExterno.vue`

Qué hace (resumen):

- Interfaz y lógica para una estructura de hashing externo usando la técnica de cambio de base como función hash primaria.
- Permite crear la estructura con parámetros de capacidad, número de dígitos por clave y la base objetivo (2..9).
- Soporta dos estrategias de resolución de colisiones: `estructura-secundaria` (bloques secundarios) y `area-colisiones` (área por bloque).
- Provee operaciones básicas: insertar, buscar y eliminar claves enteras positivas; exportar / importar la estructura en JSON.
- Visualiza la estructura en forma de bloques con áreas principales y secundarias o con áreas de colisiones según la configuración.

Algoritmos y funcionalidades principales

Creación y visualización de la estructura

1. **Crear estructura**: el usuario especifica `capacidad`, `digitosClave` y `base`. Se valida la entrada mínima (`capacidad >= 10`, `digitosClave >= 1`, `base` entre 2 y 9).
2. **Cálculo de parámetros**: se calcula `elementosPorBloque = ⌊√capacidad⌋` y `numeroBloques = ceil(capacidad / elementosPorBloque)`.
3. **Estructuras internas**:
   - `estructura`: array de bloques principales, cada uno con `elementosPorBloque` posiciones.
   - `estructuraSecundaria`: array de bloques secundarios (igual tamaño que principal), activada según resolución.
   - `areaColisiones`: array de áreas de colisiones por bloque (igual tamaño que bloque principal), usadas si `resolucionColisiones === 'area-colisiones'`.
4. **Visualización**: el template muestra bloques con `Área Principal` y `Área de Colisiones`, o muestra la `Estructura Secundaria` cuando corresponde; destaca el bloque actualmente buscado.

Gestión básica

1. **Insertar**: usa `hashCambioBaseBloque(clave)` (implementada en `utils/funciones.ts` como `HashCambioBase`) para decidir el bloque. Inserta en la primera posición libre del bloque principal; si está lleno, inserta en `areaColisiones` o en `estructuraSecundaria` según la estrategia.
2. **Buscar**: calcula el mismo hash para localizar el bloque y realiza búsqueda lineal en: bloque principal → área de colisiones → bloque secundario (si visible).
3. **Eliminar**: busca el elemento como en `buscar` y lo elimina si se encuentra, decrementando `elementosInsertados`.

Importación y exportación

1. **Exportar**: serializa la estructura actual (bloques principales, secundarios y áreas de colisiones) y permite descargarla como JSON.
2. **Importar**: permite abrir un JSON y reconstruir la estructura (con validaciones básicas).

Parámetros de configuración

- `capacidad`: número total de elementos que la estructura puede alojar (mínimo 10).
- `digitosClave`: cantidad de dígitos por clave (usado para validaciones y en la función hash si aplica).
- `base`: base para el cambio de base (2..9).
- `resolucionColisiones`: estrategia (`estructura-secundaria` o `area-colisiones`).

Manejo de errores y validaciones

- Validaciones en entrada: capacidad mínima, dígitos por clave, rango de base.
- Mensajes de error informativos (`errorMessage`) para casos como capacidad llena, duplicados, arias de colisión completas, índices inválidos.

Algoritmos y detalles de implementación

1. **Hash (Cambio de base)**: delega en `funciones.HashCambioBase(clave, numeroBloques, base)` y valida el resultado; si falla, usa `clave % numeroBloques` como respaldo.
2. **Búsqueda e inserción**: operaciones basadas en localización por bloque y búsqueda lineal dentro del bloque y sus áreas secundarias.
3. **Visualización optimizada**: `getDisplayIndicesForBlock` limita la cantidad de elementos mostrados cuando el bloque es grande (>20), mostrando primeros, últimos y ocupados.

Notas finales

- La vista combina interacción y visualización para facilitar la enseñanza del comportamiento de hashing externo con cambio de base.
- Para estructuras grandes, la visualización omite posiciones vacías no relevantes para claridad.

HashCuadradoExterno.vue

Archivos Creados

1. HashCuadradoExterno.vue

Ubicación: `/frontend/src/views/external-searches/hash/HashCuadradoExterno.vue`

Qué hace (resumen):

- Implementa una estructura de hashing externo usando la función `HashCuadrado` (algoritmo de cuadrados mediales u otra variante definida en `utils/funciones.ts`).
- Estructura y opciones de colisión análogas al módulo de cambio de base: `estructura-secundaria` o `area-colisiones`.
- Proporciona insertar, buscar, eliminar, importar y exportar la estructura.

Algoritmos y funcionalidades principales

Creación y visualización de la estructura

1. **Crear estructura**: recibe `capacidad` y `digitosClave` y valida (mínimo 10, al menos 1 dígito por clave).
2. **Parámetros calculados**: `elementosPorBloque = ⌊√capacidad⌋`, `numeroBloques = ceil(capacidad / elementosPorBloque)`.
3. **Estructuras internas**: igual que en `HashCambioBaseExterno.vue` — bloques principales, secundarios y áreas de colisiones.

Gestión básica (por bloque)

1. **Insertar**: `hashCuadradoBloque(clave)` llama a `funciones.HashCuadrado(clave, numeroBloques)` y usa módulo como respaldo si el hash es inválido.
2. **Buscar**: mismo flujo de búsqueda lineal en bloque principal → área de colisiones → estructura secundaria.
3. **Eliminar**: elimina el elemento localizado y actualiza contadores.

Importación y exportación

- Igual que en el módulo de cambio de base: import/export JSON, con validación de forma mínima.

Parámetros de configuración

- `capacidad`, `digitosClave`, `resolucionColisiones`.

Manejo de errores y validaciones

- Mensajes de error para capacidad llena, duplicados, zonas llenas, etc.

Algoritmos y detalles de implementación

1. **Hash Cuadrado**: `HashCuadrado(clave, numeroBloques)` en `utils/funciones.ts` es la función clave; la implementación del componente la valida y aplica un fallback por módulo.
2. **Inserción con resolución de colisión**: idéntica a `HashCambioBaseExterno.vue` en cuanto a comportamiento (área de colisiones o estructura secundaria).

Notas finales

- Estructura y UX prácticamente iguales a la variante de cambio de base; la diferencia radica en la función hash usada.

HashModuloExterno.vue

Archivos Creados

1. HashModuloExterno.vue

Ubicación: `/frontend/src/views/external-searches/hash/HashModuloExterno.vue`

Qué hace (resumen):

- Implementa la variante de hashing externo usando la función por **módulo** como función hash primaria.
- Permite crear la estructura con `capacidad`, `digitosClave` y elegir estrategia de resolución de colisiones (`estructura-secundaria` o `area-colisiones`).
- Proporciona operaciones: insertar, buscar, eliminar, importar y exportar la estructura en formato JSON.
- Visualiza bloques principales y, según la configuración, áreas de colisiones o estructura secundaria.

Algoritmos y funcionalidades principales

Creación y visualización de la estructura

1. **Crear estructura**: valida `capacidad >= 10` y `digitosClave >= 1`, calcula `elementosPorBloque = ⌊√capacidad⌋` y `numeroBloques = ceil(capacidad / elementosPorBloque)`.
2. **Estructuras internas**: mantiene `estructura` (bloques principales), `estructuraSecundaria` y `areaColisiones` con tamaño `elementosPorBloque` por bloque.
3. **Visualización**: muestra el bloque actual con `Área Principal` y `Área de Colisiones` o la `Estructura Secundaria` según `resolucionColisiones`.

Gestión básica

1. **Insertar**: usa `funciones.HashModulo(clave, numeroBloques)` (fallback a `clave % numeroBloques` si es inválido) para ubicar el bloque. Inserta en la primera posición libre del bloque principal; si está lleno, usa la estrategia seleccionada.
2. **Buscar**: calcula el hash por módulo y realiza búsqueda lineal en bloque principal → área de colisiones → bloque secundario (si está visible).
3. **Eliminar**: localiza el elemento y lo elimina, actualizando contadores.

Importación y exportación

1. **Exportar**: serializa `estructura`, `estructuraSecundaria`, `areaColisiones` y meta-config en un objeto con `type: 'hash-modulo-externo'` y descarga JSON.
2. **Importar**: valida la forma con `validateExternalHashImport`, verifica `type` y `config.funcionHash === 'modulo'`, y reconstruye las estructuras.

Parámetros de configuración

- `capacidad`, `digitosClave`, `resolucionColisiones`.

Manejo de errores y validaciones

- Mensajes informativos para capacidad insuficiente, duplicados, áreas llenas, índices inválidos y errores de importación.

Algoritmos y detalles de implementación

1. **Hash Módulo**: delega en `funciones.HashModulo` y aplica fallback `clave % numeroBloques` cuando procede.
2. **Inserción/búsqueda**: estrategias idénticas a otras variantes de hash externo del proyecto; comportamiento por bloque y búsqueda lineal.

Notas finales

- Consistencia UX con los otros componentes hash externos; export/import usa utilidades comunes en `utils/importExportUtils.ts`.

HashPlegamientoExterno.vue

Archivos Creados

1. HashPlegamientoExterno.vue

Ubicación: `/frontend/src/views/external-searches/hash/HashPlegamientoExterno.vue`

Qué hace (resumen):

- Implementa hashing externo usando la técnica de **plegamiento** como función hash principal.
- Permite parámetros `capacidad`, `digitosClave` y estrategia de colisiones (`estructura-secundaria` o `area-colisiones`).
- Provee insertar, buscar, eliminar, importar y exportar; visualiza bloques y áreas secundarias.

Algoritmos y funcionalidades principales

Creación y visualización de la estructura

1. **Crear estructura**: validaciones idénticas (capacidad mínima y dígitos), cálculo de `elementosPorBloque` y `numeroBloques`.
2. **Estructuras internas**: `estructura`, `estructuraSecundaria` y `areaColisiones` con el mismo layout que otras variantes.
3. **Visualización**: bloques con `Área Principal` y `Área de Colisiones` o `Estructura Secundaria`.

Gestión básica

1. **Insertar**: `funciones.HashPlegamiento(clave, numeroBloques)` determina el bloque; si falla, fallback `clave % numeroBloques`.
2. **Buscar**: búsqueda lineal en el bloque determinado y sus áreas de colisión/estructuras secundarias.
3. **Eliminar**: elimina y ajusta contadores.

Importación y exportación

1. **Exportar**: empaqueta datos con `type: 'hash-plegamiento-externo'` y descarga.
2. **Importar**: valida formato y `config.funcionHash === 'plegamiento'` antes de reconstruir.

Parámetros de configuración

- `capacidad`, `digitosClave`, `resolucionColisiones`.

Manejo de errores y validaciones

- Mensajes para capacidad llena, duplicados, áreas llenas, archivo inválido, etc.

Algoritmos y detalles de implementación

1. **Hash Plegamiento**: función en `utils/funciones.ts` usada con fallback por módulo si es necesario.
2. **Insert/Busqueda**: mismas estrategias por bloque que el resto de variantes.

Notas finales

- UI y comportamiento coherentes con los demás componentes de hashing externo; export/import centralizado.

HashTruncamientoExterno.vue

Archivos Creados

1. HashTruncamientoExterno.vue

Ubicación: `/frontend/src/views/external-searches/hash/HashTruncamientoExterno.vue`

Qué hace (resumen):

- Implementa hashing externo usando **truncamiento** como función hash principal.

- Permite `capacidad`, `digitosClave` y elegir cómo resolver colisiones.
- Provee insertar, buscar, eliminar, importar y exportar la estructura.

Algoritmos y funcionalidades principales

Creación y visualización de la estructura

1. **Crear estructura**: mismo proceso de validación y cálculo de parámetros (`elementosPorBloque`, `numeroBloques`).
2. **Estructuras internas**: bloques principales, secundarios y áreas de colisiones con tamaño `elementosPorBloque`.
3. **Visualización**: idéntica a otras variantes, con resaltado de bloques y visualización de áreas.

Gestión básica

1. **Insertar**: `funciones.HashTruncamiento(clave, numeroBloques)` con fallback a `clave % numeroBloques` si fuera inválida.
2. **Buscar**: búsqueda lineal en bloque principal → área de colisiones → bloque secundario.
3. **Eliminar**: elimina y actualiza contadores.

Importación y exportación

1. **Exportar**: `type: 'hash-truncamiento-externo'` en el JSON exportado.
2. **Importar**: valida la forma y acepta solo archivos con la función `truncamiento` declarada.

Parámetros de configuración

- `capacidad`, `digitosClave`, `resolucionColisiones`.

Manejo de errores y validaciones

- Manejo de errores consistente: capacidad insuficiente, duplicados, import inválido, etc.

Algoritmos y detalles de implementación

1. **Hash Truncamiento**: usada desde `utils/funciones.ts` con fallback por módulo.
2. **Inserción y resolución de colisiones**: comportamiento idéntico a otras implementaciones externas — primero bloque principal, luego área/estructura secundaria.

Notas finales

- Mantiene coherencia con la familia de componentes de hashing externo; comparte utilidades de import/export.

ArbolesExpansion.vue

Archivos Creados

1. ArbolesExpansion.vue

Ubicación: `/frontend/src/views/grafos/ArbolesExpansion.vue`

Qué hace (resumen):

- Permite crear un grafo ponderado (dirigido o no dirigido), añadir aristas con peso y calcular árboles de expansión mínimo y máximo mediante una variante de Kruskal.
- Visualiza el grafo con `vis-network`, muestra conjuntos de ramas (T) y cuerdas (C), y permite generar el árbol complemento (T').

Algoritmos y funcionalidades principales

Creación de grafo y visualización

1. **Crear grafo**: el usuario define `cantidadNodos` y `esDirigido`; el componente genera nodos numerados y prepara la visualización.
2. **Visualización**: usa `vis-network` para renderizar nodos y aristas con estilos, tooltips y física para distribución.

Gestión de aristas

1. **Agregar aristas**: entrada flexible (`"12"` o `"1 2"`) y validación de nodos, ausencia de bucles y duplicados.
2. **Actualizar visualización**: mantiene `nodesDataSet` y `edgesDataSet` para sincronizar la vista.

Árbol de expansión (Kruskal)

1. **Preparación**: guarda las aristas originales y ordena por peso.
2. **Algoritmo**: variante de Kruskal usando conjuntos representados como `Map<number, Set<number>>` para mantener componentes conectadas y evitar ciclos.
3. **Resultado**: construye `ramasArbol` con las aristas del árbol y `cuerdasArbol` con las aristas restantes; actualiza el grafo mostrado con las aristas del árbol.

Árbol complemento

1. **Generar complemento**: a partir de las `cuerdasArbol` crea una visualización separada (T') mostrando únicamente las cuerdas como aristas entre los mismos nodos.

Parámetros y validaciones

- `cantidadNodos`, `esDirigido`, `peso` por arista (obligatorio al insertar).
- Validaciones: formato de entrada de aristas, existencia de nodos, no duplicados, no bucles y peso obligatorio.

Notas finales

- Implementación orientada a la enseñanza: visualiza cómo Kruskal selecciona aristas y diferencia ramas/cuerdas.
- Código maneja tanto árbol mínimo (orden ascendente) como máximo (orden descendente) usando la misma lógica de conjuntos.

Matrices.vue

Archivos Creados

1. Matrices.vue

Ubicación: `/frontend/src/views/grafos/Matrices.vue`

Qué hace (resumen):

- Provee una vista general para operaciones en grafos con énfasis en representaciones matriciales: matrices de adyacencia (nodos y aristas) e incidencia.
- Permite importar y exportar grafos, crear nodos/aristas, insertar/eliminar nodos, y visualizar matrices calculadas dinámicamente.

Algoritmos y funcionalidades principales

Creación e importación

1. **Crear grafo**: configurar `cantidadNodos`, `esDirigido` y `esPonderado` y generar nodos numerados.
2. **Importar**: admite JSON con `version: '1.0'` y esquema de `GrafoJSON` validado por `validarFormatoJSON`.

Matrices dinámicas

1. **Matriz de adyacencia (nodos)**: `adjacencyMatrixNodes` construye una matriz n×n donde 1 indica adyacencia entre nodos y 0 no adyacencia; la diagonal es 0.
2. **Matriz de adyacencia (aristas)**: `adjacencyMatrixEdges` construye una matriz m×m indicando si dos aristas comparten un extremo.
3. **Matriz de incidencia**: `incidenceMatrix` (n×m) indica participación de nodos en aristas; maneja grafos dirigidos con 1 / -1 según la dirección.

Operaciones sobre el grafo

1. **Añadir arista**: parsing flexible de entrada, validación de existencia/duplicado, y opción de peso si `esPonderado`.
2. **Insertar/Eliminar nodo**: insertar crea un id nuevo (máximo + 1), eliminar borra nodos y todas sus aristas asociadas.
3. **Borrar arista**: identifica por notación `12` y la elimina.

Visualización y UX

- Paneles desplazables con control `matrixMaxHeight` para limitar la altura de las tablas y mantener la legibilidad.
- Render optimizado para evitar operaciones costosas cuando no hay datos.

Notas finales

- Módulo útil como base para cálculos posteriores (algoritmos de camino, análisis de conectividad, etc.).

---

matricesCircuitos.vue

Archivos Creados

1. matricesCircuitos.vue

Ubicación: `/frontend/src/views/grafos/matricesCircuitos.vue`

Qué hace (resumen):

- Extiende la vista de matrices con cálculo exhaustivo de circuitos y cortes en grafos ponderados, además de matrices asociadas (matriz de circuitos, matriz de cortes) y sus versiones fundamentales.
- Calcula todos los circuitos simples, los circuitos fundamentales respecto a un árbol (ej. árbol de expansión máximo), y construye las matrices que representan participación de aristas en cada circuito/corte.

Algoritmos y funcionalidades principales

Circuitos

1. **Enumeración de circuitos**: genera `todosLosCircuitos` buscando recorridos cerrados simples. El componente mantiene representaciones de circuitos como listas de aristas.
2. **Circuitos fundamentales**: a partir de un árbol de expansión (usualmente máximo en este componente) calcula los circuitos fundamentales asociados a cada cuerda.
3. **Matriz de circuitos**: cada fila corresponde a un circuito y las columnas a aristas; entradas en {-1,0,1} indican orientación y participación.

Cortes

1. **Enumeración de cortes**: `todosLosCortes` almacena subconjuntos de aristas que desconectan el grafo; se incluye comprobación de minimalidad.
2. **Cortes fundamentales**: calculados respecto a un árbol (ramas) y listados en `cortesFundamentales`.
3. **Matriz de cortes**: matiza presencia (1) o ausencia (0) de arista en cada corte.

UX y representación

- Paneles dedicados muestran listados de circuitos, cortes y matrices de forma tabular con clases que resaltan entradas positivas/negativas.
- Soporta exportación/guardado del grafo actual.

Notas finales

- Implementación combinatoria y costosa en N; orientada a visualización y enseñanza más que a uso en grafos muy grandes.

OperacionesDosGrafos.vue

Archivos Creados

1. OperacionesDosGrafos.vue

Ubicación: `/frontend/src/views/grafos/OperacionesDosGrafos.vue`

Qué hace (resumen):

- Permite crear y manipular múltiples grafos (N grafos), y ejecutar operaciones entre grafos como unión, intersección, suma, suma anillo, producto cartesiano, producto tensorial y composición.
- Visualiza cada grafo por separado usando `vis-network` y muestra el grafo resultado de la operación solicitada.

Algoritmos y funcionalidades principales

Creación y selección

1. **Crear N grafos**: el usuario indica `cantidadGrafos` y la lista `byGraph` con el número de nodos por grafo; cada grafo se crea con nodos etiquetados (p. ej. `1A`, `2A` para G1) para evitar colisiones de ids.
2. **Selector**: botones para elegir qué grafo modificar; operaciones de añadir/arista/borrar aplican al grafo seleccionado.

Operaciones entre grafos

1. **Unión**: reúne vértices y aristas de todos los grafos en una sola estructura resultado.
2. **Intersección**: conserva solo vértices/aristas comunes según la representación usada.
3. **Suma / Suma Anillo**: operaciones que combinan aristas y/o índices según la definición académica implementada.
4. **Producto Cartesiano / Tensorial / Composición**: transforma vértices y aristas según fórmula matemática; componente crea el grafo resultado y lo visualiza.

Visualización de resultado

- Genera un `grafoResultado` y lo renderiza en un `Network` separado; además muestra la representación en teoría de conjuntos (V, A) del resultado.

Notas finales

- Componente diseñado para comparar y experimentar con operaciones algebraicas entre grafos; incluye import/export de múltiples grafos.

OperacionesUnGrafo.vue

Archivos Creados

1. OperacionesUnGrafo.vue

Ubicación: `/frontend/src/views/grafos/OperacionesUnGrafo.vue`

Qué hace (resumen):

- Contiene operaciones avanzadas sobre un solo grafo: fusión de vértices, contracción de aristas, generación de grafo línea, complemento del grafo, y transformaciones que producen grafos derivados.

- Ofrece controles para insertar/eliminar nodos y aristas, fusionar vértices y contraer aristas, y botones para generar/revertir transformaciones (línea/complemento).

Algoritmos y funcionalidades principales

Transformaciones

1. **Fusionar vértices**: toma una entrada (ej: "12") y combina los dos vértices en uno solo, reasignando aristas y eliminando duplicados o bucles según la lógica implementada.
2. **Contraer arista**: reemplaza una arista por la fusión de sus extremos (contracción), preservando o eliminando bucles según la semántica del componente.
3. **Grafo línea**: construye el grafo línea (line graph) donde cada arista del grafo original se convierte en un vértice; la componente mantiene la capacidad de revertir la operación.
4. **Complemento**: genera el complemento del grafo (mismos nodos, aristas invertidas respecto a grafo completo) y permite revertirlo.

Seguridad y UX

- Deshabilita reset/operaciones destructivas mientras una transformación como grafo-línea o complemento está aplicada; ofrece botones de reversión y confirmaciones para reset.

Notas finales

- Módulo muy útil para experimentar con transformaciones de grafos y para demostrar relaciones entre estructuras derivadas.

ResiduoHuffman.vue

Archivos Creados

1. ResiduoHuffman.vue

Ubicación: `/frontend/src/views/ResiduoHuffman.vue`

Qué hace (resumen):

- Construye un árbol de Huffman a partir de un texto de entrada, calcula frecuencias de caracteres, genera códigos binarios prefijos y permite codificar/decodificar cadenas.
- Muestra la tabla de frecuencias, la longitud ponderada L (Σ frecuencia × longitud código), y ofrece exportación/importación del árbol y resultados.

Algoritmos y funcionalidades principales

Construcción del árbol y códigos

1. **Cálculo de frecuencias**: cuenta ocurrencias de cada carácter en la cadena de entrada; produce una lista de pares (carácter, frecuencia).
2. **Construcción de Huffman**: usa un algoritmo de prioridad (cola mínima) para combinar nodos de menor frecuencia hasta formar el árbol completo; genera `HuffmanResult` con `tree`, `frequencies` y

`codes`.

3. **Generación de códigos**: recorre el árbol asignando códigos binarios a cada hoja (carácter) y construye un mapa `codes` para codificar/decodificar.

Codificación y recuperación

1. **Codificar**: `encodeText(original, codes)` transforma la cadena original en una secuencia binaria según los códigos calculados.
2. **Recuperar**: dada la estructura `tree` se puede decodificar la secuencia binaria para recuperar el texto original (operación disponible en la UI después de construir el árbol).

Visualización y métricas

1. **Tabla de frecuencias**: muestra caracteres, frecuencias y su código asignado; se ordena por frecuencia descendente.
2. **Longitud L**: calcula Σ(frecuencia × longitud código) y la muestra como métrica de eficiencia del código.
3. **Vista gráfica**: renderiza el árbol de Huffman con `vis-network`, mostrando hojas e internos y etiquetando aristas con 0/1.

Import / Export

1. **Exportar**: empaqueta `type: 'huffman'`, `frequencies`, `tree` y `config.originalText` en JSON descargable.
2. **Importar**: valida `type === 'huffman'`, reconstruye `huffmanResult`, restaura `inputText` y muestra la estructura importada.

Parámetros y validaciones

- `inputText`: cadena de texto de entrada (no vacía para construir el árbol).
- Validaciones: comprueba texto no vacío al construir, valida formato del JSON al importar y habilita/deshabilita botones según estado (p. ej. exportar solo cuando hay `huffmanResult`).

Notas finales

- Componente didáctico que permite experimentar con compresión por códigos prefijos; incluye métricas y opciones de export/import para reproducir ejercicios.
- La visualización y la tabla de frecuencias ayudan a entender la relación entre frecuencia y longitud de código.

ResiduoMultiple.vue

Archivos Creados

1. ResiduoMultiple.vue

Ubicación: `/frontend/src/views/ResiduoMultiple.vue`

Qué hace (resumen):

- Implementa una estructura de residuos múltiples (probablemente un árbol o estructura basada en módulos/árboles combinados) usada para indexar claves por su residuo en una base determinada, con soporte para codificaciones tipo Huffman/varias longitudes.
- Permite insertar, buscar y eliminar entradas, además de importar/exportar la estructura y visualizarla para fines pedagógicos.

Algoritmos y funcionalidades principales

Estructura y operaciones

1. **Modelo de residuo múltiple**: la estructura agrupa claves según residuos calculados por funciones auxiliares en `utils` y mantiene buckets o subárboles por clase de residuo.
2. **Inserción/Búsqueda/Eliminación**: operaciones estándar que calculan la clase de residuo, navegan al bucket/árbol correspondiente y aplican la operación local.

Visualización y UX

1. **Paneles por residuo**: cada residuo se muestra como una cubeta o sub-árbol con entradas visibles y métricas (ocupación, colisiones).
2. **Import/Export**: serializa la estructura completa con metadatos de configuración.

Parámetros y validaciones

- `base` o `modulo` usado para calcular residuos, `bucketSize` y opciones de visualización.
- Validaciones: formatos de claves, comprobación de integridad tras importación y mensajes de error informativos.

Notas finales

- Módulo orientado a experimentar con particionamiento por residuos y comparar rendimiento/ocupación entre diferentes bases y configuraciones de buckets.

ResiduoSimple.vue

Archivos Creados

1. ResiduoSimple.vue

Ubicación: `/frontend/src/views/ResiduoSimple.vue`

Qué hace (resumen):

- Implementa la versión sencilla de la familia de residuos (tabla o arreglo indexado por `k mod m`) con operaciones básicas de insertar, buscar, eliminar y mostrar estadísticas de carga.
- Ideal para comparar con versiones más complejas (ResiduoMultiple, Huffman) y para demostraciones de comportamiento en colisiones.

Algoritmos y funcionalidades principales

Hash simple por módulo

1. **Función base**: `h(k) = k mod m` o equivalentes para claves no numéricas (con m configurable).
2. **Resolución de colisiones**: opciones simples como encadenamiento o sondeo; interfaz permite cambiar estrategia y observar métricas.

Import/Export y validaciones

1. **Exportar/Importar**: JSON con `type: 'residuo-simple'`, configuración y contenido de la tabla.
2. **Validaciones**: tamaño de tabla, tipos de claves y comprobación de límites de carga.

Notas finales

- Componente didáctico enfocado en enseñar la idea básica de particionado por residuo y técnicas simples de resolución de colisiones.

Notas Adicionales

- Aplicativo construido a partir del framework Vue con implementacion en Typescript y manejo de SPA mediante routing interno del framework.