

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

Administración Eficiente de Memoria para Bitvectors Dinámicos Adaptativos Multiarios

PROPUESTA DE TEMA DE MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

Cristobal Ignacio Puentes Bazaes

MODALIDAD:
Memoria

PROFESOR GUÍA:
Gonzalo Navarro

SANTIAGO DE CHILE
2026

1. Introducción

Los *bitvectors* son una estructura de datos que representa un arreglo de bits (0 y 1), ampliamente utilizada debido a su alta eficiencia en tiempo y espacio. En su versión estática, existen implementaciones que soportan las siguientes operaciones en tiempo constante $O(1)$ utilizando $n + o(n)$ bits de espacio, lo cual es asintóticamente óptimo [1]:

- `access(B, i)`: Recupera el bit $B[i]$.
- `rank`$_b$`(B, i)`: Cuenta cuántas veces ocurre el bit $b \in \{0, 1\}$ en el prefijo $B[1 \dots i]$.
- `select`$_b$`(B, j)`: Devuelve la posición de la j-ésima ocurrencia del bit $b \in \{0, 1\}$ en B.

Gracias a estas garantías, los *bitvectors* se han convertido en el núcleo de numerosas estructuras de datos compactas, tales como árboles cardinales, grafos, arreglos de sufijos compactos y representaciones de secuencias [2].

Por otro lado, existen versiones dinámicas que permiten modificar el contenido y la longitud del vector, añadiendo las siguientes operaciones:

- `write(B, i, v)`: Establece $B[i] \leftarrow v$.
- `insert(B, i, v)`: Inserta el bit v en la posición i, desplazando los bits siguientes.
- `delete(B, i)`: Elimina el bit $B[i]$, desplazando los bits siguientes.

Esta variante habilita escenarios de aplicación mucho más amplios. Sin embargo, la cota inferior teórica en el modelo *cell-probe* para estas operaciones es $\Omega\left(\frac{\log n}{\log \log n}\right)$, lo que implica que la mutabilidad vuelve a la estructura un orden de magnitud más lenta en comparación con la versión estática.

No obstante, en la gran mayoría de las aplicaciones reales que emplean *bitvectors*, la frecuencia de consultas supera ampliamente a la de modificaciones. Dado que las operaciones de actualización son las responsables de elevar el costo temporal, es posible definir q como la razón entre la cantidad de consultas y la cantidad de actualizaciones que experimenta la estructura.

A partir de este enfoque adaptativo, se demostró teóricamente que es posible alcanzar una cota de $O\left(\frac{\log(n/q)}{\log \log n}\right)$ amortizada mediante una jerarquía basada en árboles multiarios balanceados [3].

Posteriormente, se implementó una variante práctica sobre un árbol binario que sigue estos mismos principios adaptativos, logrando una cota de $O(\log(n/q))$ amortizada y demostrando que esta formulación ofrece ventajas significativas frente a implementaciones dinámicas del estado del arte, las cuales suelen pagar $O(\log n)$ por operación [2].

A pesar de esto, cuando n es suficientemente grande (cercano a 2^{30}), esta implementación binaria es superada por soluciones multiarias clásicas con cota $O\left(\frac{\log n}{\log \log n}\right)$ que son insensibles a q (como el DPR). La principal hipótesis ante este comportamiento es que el árbol

binario presenta una peor localidad de referencia en la jerarquía de memoria caché frente a un árbol multiario, motivando así el desarrollo de una variante multiaria adaptativa [2].

Si bien existe una implementación preliminar de un árbol multiario adaptativo, esta sufre de una severa fragmentación de memoria y sobrecarga en la gestión de memoria dinámica al mutar sus nodos. El objetivo de esta memoria es resolver este problema de administración de espacio y memoria, evaluando exhaustivamente el desempeño de la nueva estructura resultante.

2. Situación Actual

El desarrollo propuesto se fundamenta directamente en los trabajos de mi profesor guía, G. Navarro [2, 3]. En el primero, se diseñó e implementó una solución práctica adaptativa en tiempo $O(\log(n/q))$, donde q representa la frecuencia relativa de consultas por cada actualización. En el segundo, se demostró analíticamente cómo una generalización multiaria alcanza la cota asintóticamente óptima.

En la variante práctica binaria, el *bitvector* se modela como un árbol balanceado por peso compuesto por dos tipos de hojas: las *dinámicas*, que almacenan una cantidad acotada de palabras de máquina y admiten modificaciones locales; y las *estáticas*, que almacenan bloques de longitud arbitraria representados mediante estructuras estáticas compactas que no admiten mutación directa. Cuando una hoja estática debe modificarse, esta debe romperse para volver a convertirse en nodos dinámicos.

Para garantizar la cota amortizada, se introducen dos operaciones estructurales:

- *Flattening*: cuando un subárbol acumula una cantidad suficiente de consultas sin haber recibido actualizaciones recientes, se recorre por completo y se colapsa (aplana) en una única hoja estática continua con soporte $O(1)$ para consultas.
- *Splitting*: cuando ocurre una actualización sobre una hoja estática, esta se divide sucesivamente a la mitad; una de las mitades se preserva como hoja estática, mientras que la otra continúa subdividiéndose recursivamente hasta aislar la región a modificar en una hoja dinámica.

Gracias a este esquema se supera la barrera de $O(\log n)$, operando en tiempo amortizado $O(\log(n/q))$, el cual converge a $O(1)$ a medida que la carga de trabajo está dominada por consultas.

Al extender este diseño a un árbol multiario, cada nodo interno pasa a tener una aridad de $\Theta(\sqrt{\log n})$ hijos. La diferencia crítica surge en el proceso de *splitting*: ante una actualización, la división de un nodo genera $\Theta(\sqrt{\log n})$ nuevas hojas estáticas por cada nivel de la recursión. Teóricamente, esto permite alcanzar la cota óptima de $O\left(\frac{\log(n/q)}{\log\log n}\right)$ amortizada [3].

Sin embargo, al trasladar este diseño a la práctica surgen importantes desafíos de gestión de memoria. Las estructuras estáticas de soporte para `rank` y `select` dependen de directorios precalculados de bloques y superbloques. En la versión binaria, cada partición divide un bloque en solo dos mitades. En cambio, en la versión multiaria, la creación de $\Theta(\sqrt{\log n})$ hojas estáticas independientes en cada paso obliga a recalcular y realocar estas estructuras auxiliares desde cero, provocando un elevado número de llamadas al alocador de memoria y un alto volumen de copias redundantes de bits.

Actualmente existe una implementación preliminar que sufre de este problema de degradación práctica, impidiendo que el árbol multiario supere a las alternativas existentes para valores grandes de n. En esta memoria se abordará dicho cuello de botella mediante una técnica de desacoplamiento basada en *búfers flotantes* e índices de desplazamiento (*offsets*).

Bajo este enfoque, al producirse un *splitting* el búfer estático original no se destruye ni se replica en múltiples sub-arreglos. El bloque de memoria original permanece "flotando" en el montículo, y las nuevas hojas estáticas hijas se crean como descriptores ligeros que contienen punteros al búfer compartido junto con los intervalos de bits que definen su rango lógico.

No obstante, esta técnica introduce desafíos algorítmicos sobre las operaciones `rank` y `select`, ya que los directorios del búfer flotante están construidos respecto al inicio global del bloque y no respecto al rango local de la hoja. Adicionalmente, el desacoplamiento introduce un problema de fragmentación y retención de memoria: con el tiempo, mutaciones sucesivas o procesos de *flattening* en otras regiones pueden dejar sub-rangos del búfer flotante sin ninguna hoja que los referencie (memoria huérfana), generando un sobreuso de memoria que compromete la compacidad teórica del sistema si no se incorpora un esquema adecuado de recolección y compactación.

3. Objetivos

Objetivo General

Diseñar e implementar un esquema de administración de memoria y espacio para hojas estáticas compartidas en la implementación actual del *bitvector* dinámico adaptativo multiario, con el fin de mitigar la fragmentación y el exceso de copias de memoria, y evaluar experimentalmente su desempeño en tiempo y espacio en comparación con la versión multiaria base sin administración, la versión binaria adaptativa y estructuras competitivas del estado del arte como el DPR.

Objetivos Específicos

1. Diseñar e implementar el mecanismo de hojas estáticas como vistas sobre búfers flotantes compartidos en el árbol multiario, adaptando los algoritmos de `rank` y `select` para operar correctamente mediante desplazamientos locales y globales en tiempo $O(1)$.
2. Diseñar e implementar una política de administración de memoria y compactación para los búfers flotantes, que identifique porciones de memoria huérfanas o con baja densidad de uso, recolecte recursos no referenciados y evite fugas de memoria (*memory leaks*).
3. Evaluar empíricamente el desempeño de la estructura mediante una batería de experimentos de rendimiento, comparando la variante propuesta frente a la versión multiaria sin administración, la versión binaria adaptativa y el DPR bajo diferentes tamaños n y razones de consulta/actualización q, midiendo tiempo de CPU, consumo pico de memoria y fragmentación.

Evaluación

El cumplimiento de los objetivos se evaluará mediante la obtención de una estructura multiaria completamente funcional, verificada mediante pruebas de corrección y calibrada para operar sin fugas de memoria ni degradación espacial excesiva. La evaluación de desempeño consistirá en someter las estructuras a cargas de trabajo sintéticas controladas variando el tamaño del *bitvector* (n desde valores moderados hasta escalas donde la memoria caché y la jerarquía de memoria sean críticas, cercanas a 2^{30} bits) y ajustando la razón de operaciones q en distintos órdenes de magnitud.

Se medirá el tiempo promedio por operación para consultas (`access`, `rank`, `select`) y actualizaciones (`insert`, `delete`), así como el uso efectivo de memoria principal (*peak resident set size*) y el espacio promedio por bit representado. El éxito del trabajo radicará en identificar con precisión bajo qué regímenes de n y q la variante multiaria con administración de espacio supera a la variante binaria y logra cerrar la brecha de rendimiento frente al DPR.

4. Solución Propuesta

La fundamentación teórica del árbol multiario adaptativo se encuentra formalizada en [3], mientras que el diseño de referencia en árbol binario y su evaluación empírica se detallan en [2]. Se dispone actualmente de una implementación en desarrollo en C++ del árbol multiario que se apoya en componentes de la biblioteca SDSL (*Succinct Data Structure Library*), la cual servirá como punto de partida y base comparativa para integrar el esquema de administración propuesto.

El software se desarrollará en C++ (estándar C++20), aprovechando el control explícito sobre la memoria y las instrucciones a nivel de palabra de máquina (*popcount*, operaciones bit a bit). La solución se centrará en reestructurar el mecanismo de *splitting* para evitar la realocación continua y las copias redundantes de sub-arreglos de bits que actualmente degradan el rendimiento de la estructura.

El principio central consiste en desacoplar el almacenamiento físico de los bits de la representación lógica de las hojas. En lugar de que cada hoja estática mantenga una copia privada e independiente de sus datos, pasará a comportarse como un descriptor o vista lógica sobre un bloque de memoria compartido (un *búfer flotante*). Al producirse una partición por actualización, el búfer estático original se mantiene en memoria sin ser destruido ni duplicado, y las nuevas hojas hijas se instancian referenciando dicho bloque común junto con la delimitación del intervalo de bits que les corresponde gobernar.

Para soportar las consultas sobre estas vistas sin perder las cotas de tiempo constante:

- $\texttt{rank}_b\texttt{(i)}$: El cálculo local se traduce a coordenadas globales del búfer flotante mediante el desplazamiento (*offset*) inicial de la hoja, evaluando la diferencia entre el rango acumulado hasta la posición consultada y el rango previo al inicio del intervalo ($\text{rank}_b(j) - \text{base_rank}$), lo que permite conservar el tiempo de consulta en $O(1)$ apoyándose en los directorios precalculados del bloque base.
- $\texttt{select}_b\texttt{(k)}$: De forma análoga, se resuelve buscando la posición global correspondiente al $(k + \text{base_rank})$-ésimo bit dentro del búfer subyacente y trasladando el resultado de vuelta a la escala local de la hoja.

El desafío principal de este diseño radica en la gestión del ciclo de vida de los búfers compartidos, ya que mutaciones sucesivas o procesos posteriores de *flattening* pueden dejar sub-rangos sin hojas activas que los referencien. Para abordar este problema de memoria huérfana y retención innecesaria de espacio, el diseño preliminar contempla un sistema de contabilidad basado en métricas de utilidad (como la relación entre bits útiles referenciados y el tamaño total del búfer, sumado a un seguimiento de referencias activas). A partir de estos indicadores se evaluarán y compararán distintas políticas de compactación:

- **Compactación orientada a umbrales de densidad:** Reubicar los rangos útiles y liberar el bloque original cuando la proporción de bits vivos caiga por debajo de un porcentaje crítico.
- **Compactación perezosa (*lazy*) o amortizada:** Postergar la consolidación y liberación de memoria hasta momentos de rebalanceo del árbol o fases de re-aplanamiento, evitando sobrecargar las operaciones de actualización frecuentes.

La selección definitiva de la política de compactación y el ajuste de sus umbrales se afinarán empíricamente durante la etapa de experimentación, evaluando el compromiso (*trade-off*) entre el uso pico de memoria y el costo temporal que introduce la recolección. Finalmente, la variante implementada se contrastará frente a la versión multiaria base sin administración, la versión binaria adaptativa y estructuras competitivas consolidadas como el DPR, analizando tiempos de respuesta por operación, comportamiento asintótico y consumo de memoria bajo diversas combinaciones de n y q.

5. Plan de Trabajo (Preliminar)

1. **Estudio y análisis del código base:** Revisión exhaustiva de las bases teóricas [2, 3] y auditoría del código C++ de la implementación multiaria existente, identificando los cuellos de botella exactos durante las fases de *splitting* y caracterizando el consumo de memoria basal.
2. **Diseño e implementación de búfers flotantes:** Modificación de la estructura de las hojas estáticas para actuar como referencias/vistas sobre búfers compartidos, junto con la reimplementación y verificación de los algoritmos de `access`, `rank` y `select` adaptados a desplazamientos de intervalo en tiempo $O(1)$.
3. **Validación funcional inicial y caracterización de fragmentación:** Pruebas de corrección lógica y medición preliminar de la retención de memoria y generación de fragmentos huérfanos generados tras secuencias de inserciones y consultas.
4. **Diseño e implementación del administrador de memoria y compactación:** Desarrollo de los esquemas de seguimiento de densidad (contadores de referencia y bits activos) e implementación de rutinas de recolección y compactación de búfers flotantes cuando su utilidad decae.
5. **Pruebas de estrés y análisis de memoria:** Verificación de ausencia de fugas de memoria (*memory leaks*) y validación de la estabilidad del uso del heap mediante herramientas como Valgrind/Massif y AddressSanitizer.
6. **Experimentación y evaluación comparativa:** Montaje y ejecución de la suite de pruebas sobre conjuntos de datos escalables hasta $n \approx 2^{30}$ bits y diversos valores de q, contrastando el rendimiento frente a la versión multiaria sin recolección, la versión binaria adaptativa y el DPR.
7. **Análisis de resultados y redacción de memoria:** Procesamiento estadístico de las métricas de tiempo y espacio, elaboración de gráficos comparativos y redacción del informe final de titulación.

Referencias

[1] Clark, D. R.: *Compact PAT trees.* Tesis de Doctorado, University of Waterloo, Canada, 1996.

[2] Navarro, G.: *Practical adaptive dynamic bitvectors.* Software: Practice and Experience, 55(9):1539–1559, 2025.

[3] Navarro, G.: *(Worst-case) Optimal Adaptive Dynamic Bitvectors.* Theory Comput Syst, 69(30), 2025.