Comparación de Memoria y Tiempo en Algoritmos de Búsqueda Top-1

Javier Jhairt López Rojas

18 de febrero de 2026

1. Introducción

Este reporte analiza el consumo de memoria y el tiempo de ejecución al resolver la búsqueda top-1 (encontrar el vector $x \in X$ que maximice el producto punto con $q \in Q$). Básicamente, comparamos qué es más eficiente en la práctica: multiplicar matrices de golpe o usar ciclos iterativos con referencias.

2. Metodología

2.1. Supuestos

Los supuestos realizados para el desarrollo de esta practica son los siguientes:

- Entrada de datos y dimensiones: Se analizarán combinaciones de $m \in \{10^3, 10^6, 10^9\}$, $n \in \{10^3, 10^6, 10^9\}$ y dimensiones $d \in \{8, 16, 32\}$.
- Normalización: Las matrices Q y X contienen datos aleatorios cuyos vectores columna están normalizados sobre la esfera unitaria, por lo que maximizar el producto punto es matemáticamente equivalente a minimizar la distancia coseno.
- Tipos de datos: En un mundo real, para evitar daños de hardware y saturación de memoria RAM (Out of Memory), se asume el uso de datos de tipo `Float32` (4 bytes) e `Int64` (8 bytes).

2.2. Herramientas y entorno

Los experimentos se realizaron en el lenguaje de programación Julia. Se utilizaron las estructuras de control nativas, la librería `LinearAlgebra` para las operaciones vectoriales,

y la macro `@time` para reportar los costos reales de ejecución y asignación de memoria. Para las gráficas se usó la librería `Plots`.

3. Implementación de Algoritmos A1 y A2

3.1. Algoritmo A1: Operaciones matriciales explícitas

Este algoritmo realiza el producto matricial completo $P = Q^T X$, calculando todos los pares posibles en un solo paso, para luego extraer el máximo de cada fila.

3.2. Algoritmo A2: Adaptación con referencias

Este algoritmo adapta la función `getmaxdot`, iterando sobre las columnas de Q y calculando los productos punto bajo demanda usando `@view` o referencias abstractas para no copiar grandes bloques de memoria.

4. Análisis de memoria necesaria

Se analiza la memoria asintótica requerida. El algoritmo A1 tiene una complejidad espacial de $O(m \times n)$ ya que debe alojar la matriz P. El algoritmo A2 tiene un costo $O(m)$ pues solo guarda los vectores de resultados `nns` y `dots`.

Cuadro 1: Memoria extra requerida para A1 vs A2 (usando Float32 e Int64).

m (cols de Q)	n (cols de X)	A1 ($O(m \times n)$)	A2 ($O(m)$)
10^3	10^3	$\approx 4\,\text{MB}$	$\approx 12\,\text{kB}$
10^3	10^6	$\approx 4\,\text{GB}$	$\approx 12\,\text{kB}$
10^3	10^9	$\approx 4\,\text{TB}$	$\approx 12\,\text{kB}$
10^6	10^3	$\approx 4\,\text{GB}$	$\approx 12\,\text{MB}$
10^6	10^6	$\approx 4\,\text{TB}$	$\approx 12\,\text{MB}$
10^6	10^9	$\approx 4\,\text{PB}$	$\approx 12\,\text{MB}$
10^9	10^3	$\approx 4\,\text{TB}$	$\approx 12\,\text{GB}$
10^9	10^6	$\approx 4\,\text{PB}$	$\approx 12\,\text{GB}$
10^9	10^9	$\approx 4\,\text{EB}$	$\approx 12\,\text{GB}$

Discusión: Es interesante observar que A1 colapsará la memoria de una computadora convencional al cruzar el umbral de $m = 10^3, n = 10^6$, mientras que A2 permanece

altamente eficiente incluso para $m = 10^9$.

5. Reporte de costos en tiempo real

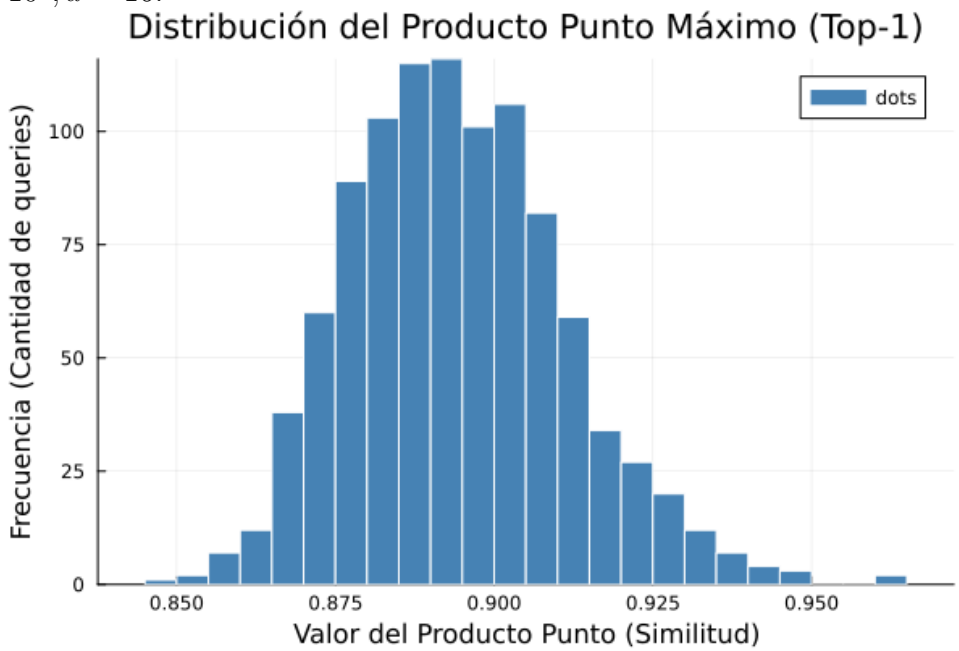
Para una corrida con los argumentos $m = 10^3$, $n = 10^6$ y $d = 16$, se reportaron los siguientes costos en consola de Julia:

- **A1**: 11.161083 seconds (1.64 k allocations: 3.725 GiB)
- **A2**: 5.875192 seconds (2.01 k allocations: 136.969 KiB)

Discusión: Podemos notar que A2 fue sustancialmente más rápido que A1. Esto sucede porque A1 se ve estrangulado por el cuello de botella físico de asignar y escribir casi 4 GB de memoria RAM. A2, al procesar cada producto punto al vuelo sin guardar la matriz entera, aprovecha la memoria caché del procesador, reduciendo drásticamente las asignaciones a solo unos 136 kB.

6. Distribución del arreglo *dots*

Figura 1: Distribución de los valores máximos del producto punto para la corrida $m = 10^3, n = 10^6, d = 16$.



Discusión: Se puede apreciar que la distribución está fuertemente sesgada a la derecha, con valores muy próximos a 1.0. Al existir un millón de candidatos en la esfera unitaria en un espacio de solo 16 dimensiones, la probabilidad matemática de encontrar un vector casi colineal es sumamente alta.

7. Conclusiones

De esta práctica nos quedan dos cosas claras. Primero, la gráfica demuestra que al buscar entre un millón de vectores en solo 16 dimensiones, casi siempre encontraremos un vecino muy similar (formando una campana centrada en ≈ 0.89). Segundo, la eficiencia de un algoritmo depende totalmente de cómo gestiona la memoria. El algoritmo A2 superó a A1 porque calcula todo al vuelo aprovechando la memoria caché, mientras que A1 se asfixia intentando guardar una matriz gigantesca e innecesaria en la RAM.

8. Referencias

- Unidad 3: Notas del curso (Introducción a propiedades y operaciones básicas de estructuras y matrices en Julia).
- Gómez Fuentes, M. C., & Cervantes Ojeda, J. (2014). Análisis de la complejidad algorítmica. En *Introducción al Análisis y al Diseño de Algoritmos*. Universidad Autónoma Metropolitana.