Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Informe Tarea 2
Tries para autocompletado

Integrantes: Fernanda Contreras
Nicolás Inostroza
María Acuña

Profesores: Gonzalo Navarro

Auxiliar: Diego Arias
Claudio Gaete

Ayudante: Vicente Oliva
Cristóbal Puentes

Fecha de entrega: 6 de noviembre de 2025
Santiago, Chile

Índice

1. Introducción

En este informe se estudia el uso de Tries para poder realizar el autocompletado de palabras, se utilizaran 2 variantes distintas de Tries, una de ellas retorna la palabra del subárbol que fue accedida más recientemente, la otra variante devuelve la palabra con mayor frecuencia de accesos.

Inicialmente se utiliza un dataset "words.txt", que contiene un total de $N = 2^{18}$ palabras, para poder medir los tiempos de inserción y la cantidad de memoria usada por el trie durante diferentes momentos de la inserción completa del dataset.

Luego, se simulara la capacidad de autocompletado del trie creado. Para esto se simulará la escritura de las distintas palabras de un texto letra por letra. La idea es poder tener una medida de cuantas letras se habría ahorrado un usuario, asumiendo que en el momento que su palabra sea correctamente autocompletada deja de escribirla y pasa a la siguiente.

Para esta simulación se utilizaran 3 datasets:
- wikipedia.txt, que contiene texto real de distintas páginas de Wikipedia
- random.txt, que usa palabras del mismo dataset de words.txt con distribución uniforme
- random_with_distribution.txt que también es generada de forma aleatoria pero con una distribución más realista de palabras.

Nuestra hipótesis es que dado un texto "realista" como lo es el dataset de Wikipedia, la variante de frecuencia sea más efectiva para el autocompletado dado el mayor uso de conectores, pronombre y verbos comunes como por ejemplo el verbo "ser" o "be" en ingles. Sin embargo al variante de tiempo, es mas adaptable a los cambios de contextos que la variante de frecuencia.

2. Desarrollo

A continuación se presentan las decisiones de implementación de algoritmos y estructuras de datos para ambas variantes de Tries.

2.1. Arquitectura

La solución se organizó en una estructura modular compuesta por los siguientes archivos:

Archivos de implementación:
- `trie.h`: Contiene la definición de las esrtuctuas `TrieNode` y `Trie`, junto con todas las operaciones fundamentales del árbol de prefijos.
- `experimentos.cpp`: Implementa los tres experimentos principales con los datasets, midiendo memoria, tiempo y capacidad de autocompletado en ambas variantes.
- `interfaz.cpp`: Proporciona una interfaz interactiva para demostrar el funcionamiento del trie, con cambio de modo para ambas variantes.

2.2. Decisiones de implementación

La implementación de los algoritmos se realizó siguiendo los pasos mencionados en el enunciado de esta tarea y las principales diferencias de implementación se mencionan a continuación:

2.2.1. Estructura de datos: TrieNode

El nodo del trie se implementó manteniendo los campos especificados en el enunciado: `parent`, `next[27]`, `priority`, `str`, `best_terminal` y `best_priority`.

2.2.2. Estructura de datos: Trie

La estructura principal del trie mantiene:

- **root:** Una referencia a la raíz, prefijo vacío $, es el punto de entrada para todas las operaciones.

- **node_count:** Es un contador que se incrementa en 1 cada vez que se crea un nodo. Esta decisión de diseño permite medir el consumo de memoria de la estructura de forma directa sin tener que hacer recorridos adicionales por el árbol.
- **access_counter:** Es un contador que se usa sólo para la variante reciente, el cual se incrementa cada vez que se accede a una palabra proporcionando un timestamp relativo que refleja el orden temporal de los accesos.

Esta estructura implementa las siguiente operaciones:

- **insert(const string& w) y descend(TrieNode* v, char c):** Ambas operaciones se implementaron siguiendo el enunciado. Insert crea nodos carácter por carácter, y descend navega por un carácter específico.
- **autocomplete(TrieNode* v):** Retorna el nodo terminal con mayor prioridad en el subárbol de `v`. La decisión de diseño acá fue implementar validaciones antes de retornar, se verifica que `best_priority >= 0`, que `best_terminal` no sea nulo, que apunte a un string válido, y que sea realmente un nodo terminal. Estas validaciones previenen accesos a memoria inválida.
- **update_priority_frequency(TrieNode* terminal) y update_priority_recent(TrieNode* terminal):** Se separaron en dos funciones distintas, una para cada variante. En frecuencia se incrementa priority en 1, mientras que en reciente se asigna el valor actual de access_counter. Esta separación evita condicionales en tiempo de ejecución y clarifica la intención del código para cada variante.
- **propagate_best(TrieNode* v):** Esta operación se agrego para realizar la propagación de cambios de prioridad hacia la raíz. La decisión de diseño acá es el almacenamiento de los valores antiguos de `best_priority` y `best_terminal`, ya que si estos no cambian después de evaluar todos los hijos, se rompe el loop de propagación. Esto evita iterar innecesariamente sobre ancestros cuando la actualización solo afecta ramas locales.
- **get_node_count():** Función auxiliar que retorna el número total de nodos creados guardado en `node_count` en la estructura `Trie`.

2.2.3. Funciones auxiliares

- **destroy_tree(TrieNode* node):** Destruye recursivamente el árbol liberando toda la memoria.
- **get_terminal(Trie& trie, const string& word):** Busca y retorna el nodo terminal correspondiente a una palabra.
- **load_words(const string& filename):** Carga palabras desde un archivo de texto a un vector.
- **create_measurement_points(long long max_val):** Genera los puntos de medición (potencias de 2) para los experimentos.
- **run_memory_experiment(const vector& words):** Mide el consumo de memoria en puntos de referencia durante la construcción del trie.
- **run_time_experiment(const vector& words):** Mide el tiempo de inserción normalizado por caracteres en grupos (16 grupos).
- **run_autocomplete_simulation(Trie& trie, const vector& text_data, const string& variant_name, const string& dataset_name):** Simula la escritura del usuario letra por letra, registrando cuántos caracteres se habrían ahorrado usando autocompletado. Implementa la lógica completa: descender por carácter, invocar autocomplete, comparar con la palabra objetivo, y actualizar las prioridades. Se ejecuta para ambas variantes y tres datasets distintos (Wikipedia, random uniforme, random con distribución realista).
- **read_words(const string& filename, vector& words, long long limit = −1):** Lee palabras desde archivos con límite opcional.

2.3. Interfaz

Se implementó una interfaz gráfica utilizando la librería **Raylib**, la cual debe descargarse para compilar y ejecutar la interfaz.

La implementación consta de la estructura `TrieInterface` que mantiene el estado del trie (`trie`), el prefijo actual(`current_prefix`), el texto acumulado (`current_text`) y la variante seleccionada (`variant`), junto con funciones que soportan la interacción:

- **get_suggestion()**: Obtiene la palabra sugerida basada en el prefijo actual.
- **accept_suggestion()**: Acepta la sugerencia y actualiza la prioridad según la variante.
- **write_prefix()**: Escribe el prefijo actual como palabra completa.
- **backspace()**: Elimina el último carácter del prefijo.
- **add_char()**: Agrega un carácter al prefijo.
- **draw_interface()**: Dibuja la interfaz gráfica con el estado actual.

La interfaz permite cambiar la variante del trie presionando la tecla `1`, escribir prefijos con autocompletado en O(1), aceptar una recomendación con TAB, eliminar carácteres con BACKSPACE, escribir el prefijo con ENTER y salir de la ventana con ESC.

3. Resultados

La obtención de los resultados fue realizada en Windows 11, con un caché L1 = 1MB, L2 = 8MB y L3 = 32MB y 32GB de RAM. Todos los experimentos realizados fueron con el conjunto de datos principal words.txt que contiene 262144 palabras en inglés. Además, para las simulaciones de autocompletado se utilizaron los datasets wikipedia.txt, random.txt y random_with_distribution.txt

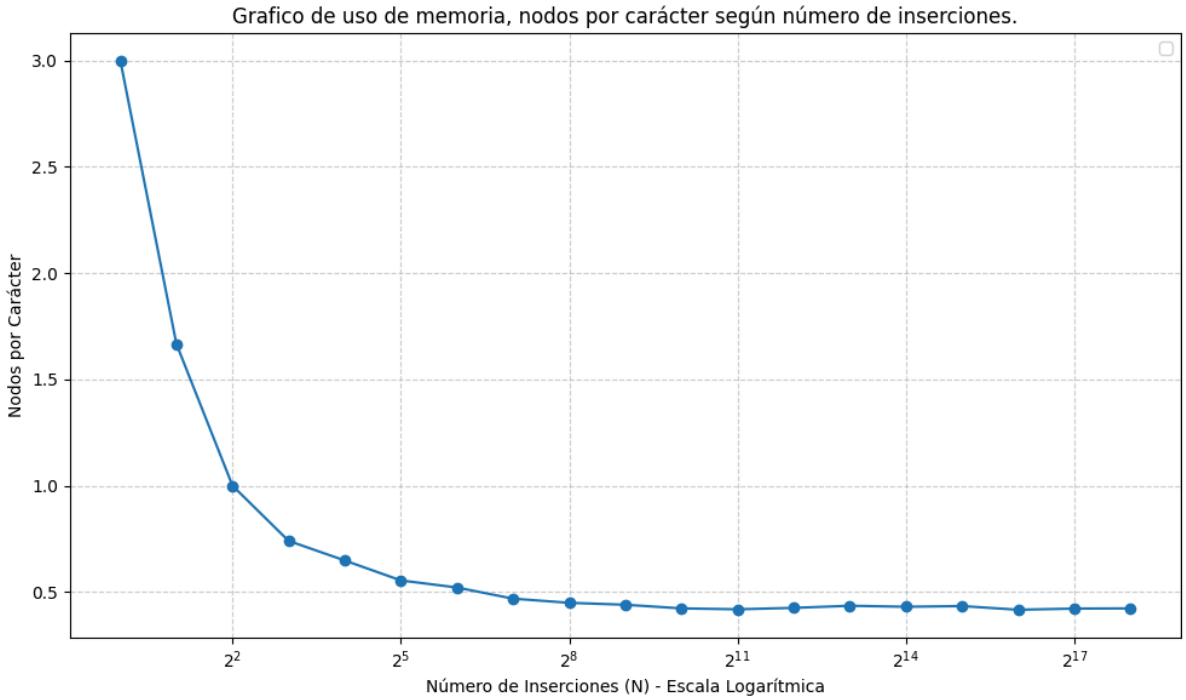
3.1. Medición de memoria (cantidad de nodos)

Durante la construcción del Trie con el dataset principal words.txt, se midió la cantidad total de nodos creados durante diferentes partes de la inserción de las palabras del dataset de words.txt.

Inserción N	Nodos totales	Caracteres totales	Nodos por carácter
1	3	1	3.000000
2	5	3	1.666667
4	9	9	1.000000
8	20	27	0.740741
16	48	74	0.648649
32	91	164	0.554878
64	205	393	0.521628
128	418	891	0.469136
256	875	1946	0.449640
512	1801	4088	0.440558
1024	3667	8660	0.423441
2048	7776	18551	0.419169
4096	15925	37396	0.425848
8192	31567	72453	0.435689

16384	65135	151096	0.431084
32768	124976	287753	0.434317
65536	247709	593891	0.417095
131072	496004	1173952	0.422508
262144	1021219	2412275	0.423343

Graficando estos datos podemos ver como varia la cantidad de nodos por carácter según la cantidad de palabras insertadas de manera mas clara.



Con esto podemos ver como la cantidad de nodos por carácter va disminuyendo de manera casi que logarítmica entre mas palabras se insertan en el Trie.

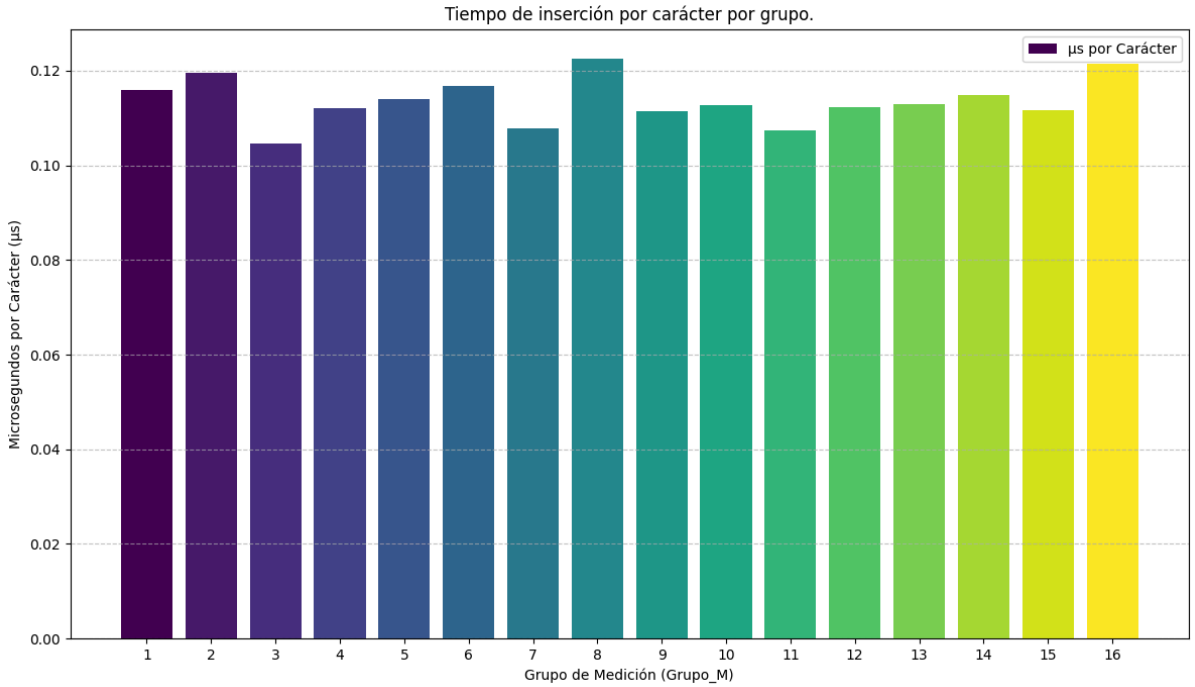
3.2. Medición de tiempos de ejecución

Por otro lado medimos la cantidad de tiempo que requiere la inserción de grupos de palabras de tamaño $\frac{N}{M}$, así como el tiempo promedio de inserción por carácter de cada grupo.

Grupos	Tiempo total (µs)	Caracteres por grupo	Tiempo por carácter (µs)
1	17523	151096	0.115973
2	16347	136657	0.119621
3	16087	153819	0.104584
4	17084	152319	0.112159
5	16642	145915	0.114053
6	16046	137352	0.116824
7	17624	163618	0.107714
8	16333	133176	0.122642

9	18218	163602	0.111356
10	17782	157876	0.112633
11	17925	166972	0.107353
12	16414	146138	0.112318
13	16120	142757	0.112919
14	17788	154819	0.114895
15	18161	162788	0.111562
16	17430	143371	0.121573

Si representamos estos datos en un grafico podemos ver la variación de tiempo de inserción por carácter en cada uno de los grupos medidos.



En este grafico de columnas podemos ver que el tiempo de inserción de caracteres por grupo se mantiene similar entre todos los grupos, con un tiempo entre 0.11 y 0.13 microsegundos.

3.3. Simulación de autocompletado

Para medir la eficiencia de autocompletado de ambas variantes de Trie usamos 3 datasets con los que simulamos la escritura carácter a carácter de cada palabra dentro del dataset.

Los datasets usados son:
- wikipedia.txt, que contiene texto real de distintas páginas de Wikipedia
- random.txt, que usa palabras del mismo dataset de words.txt con distribución uniforme
- random_with_distribution.txt que también es generada de forma aleatoria pero con una distribución más realista de palabras.

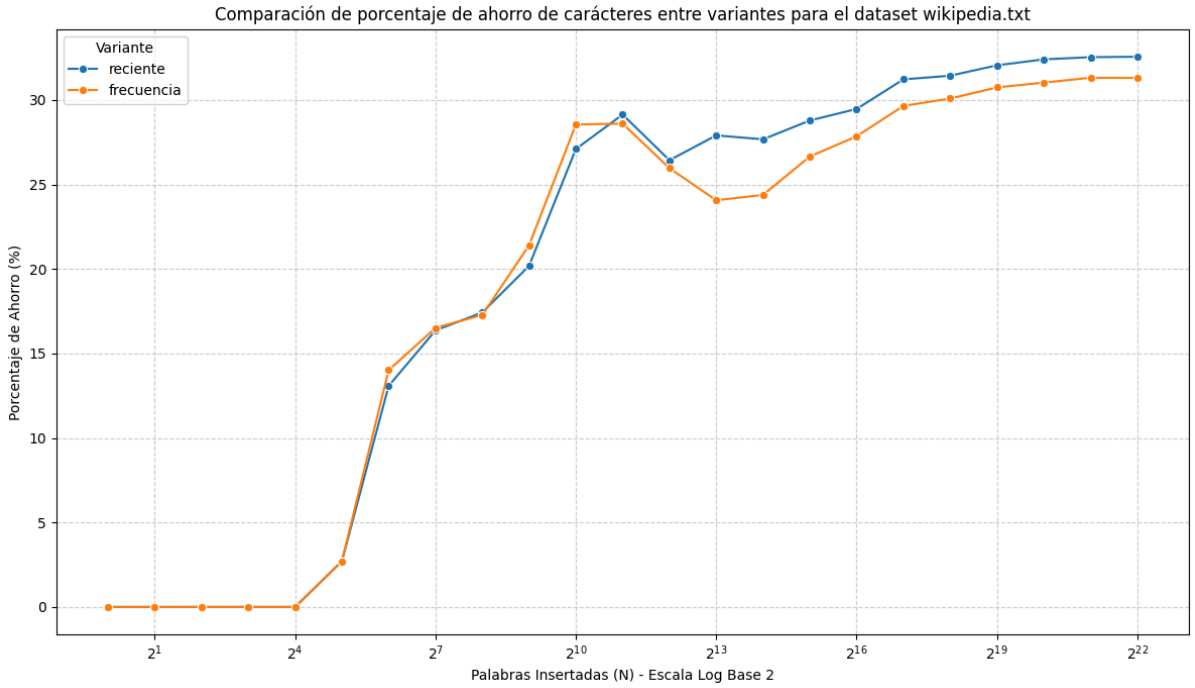
3.3.1. Porcentaje de ahorro

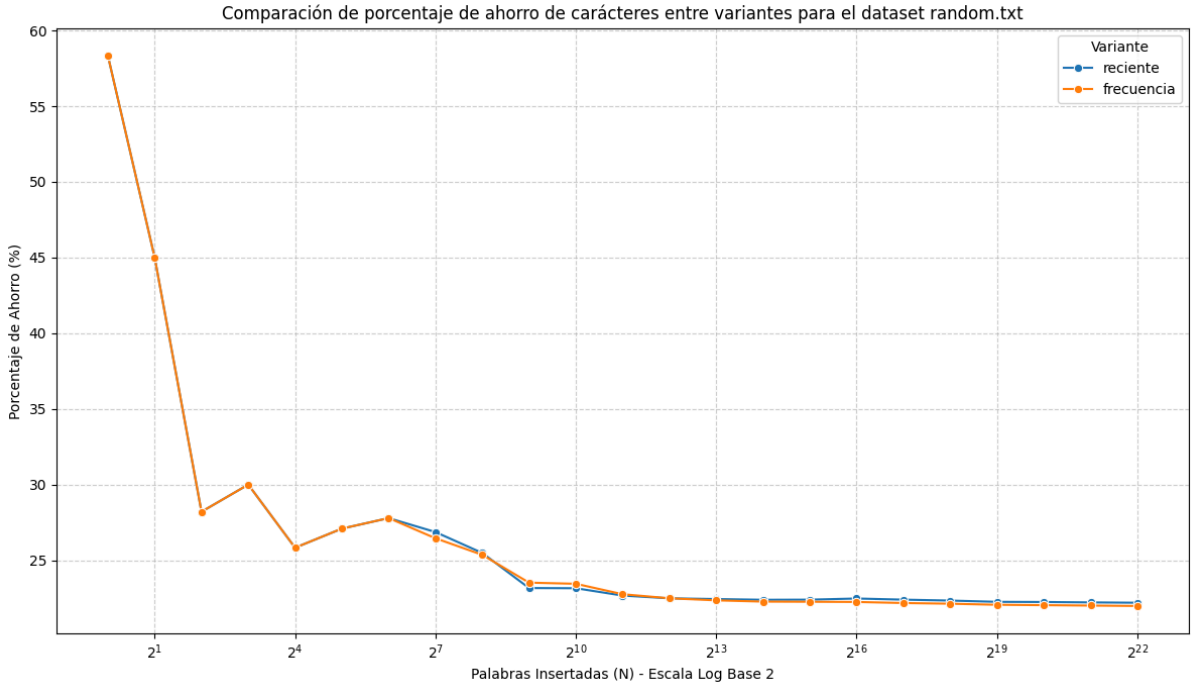
Para medir el porcentaje de ahorro se mide, en porcentaje, la diferencia entre la cantidad de caracteres escritos y la cantidad total de caracteres. Esta medida se hace para cada variante en cada uno de los dataset obteniendo los siguientes resultados finales.

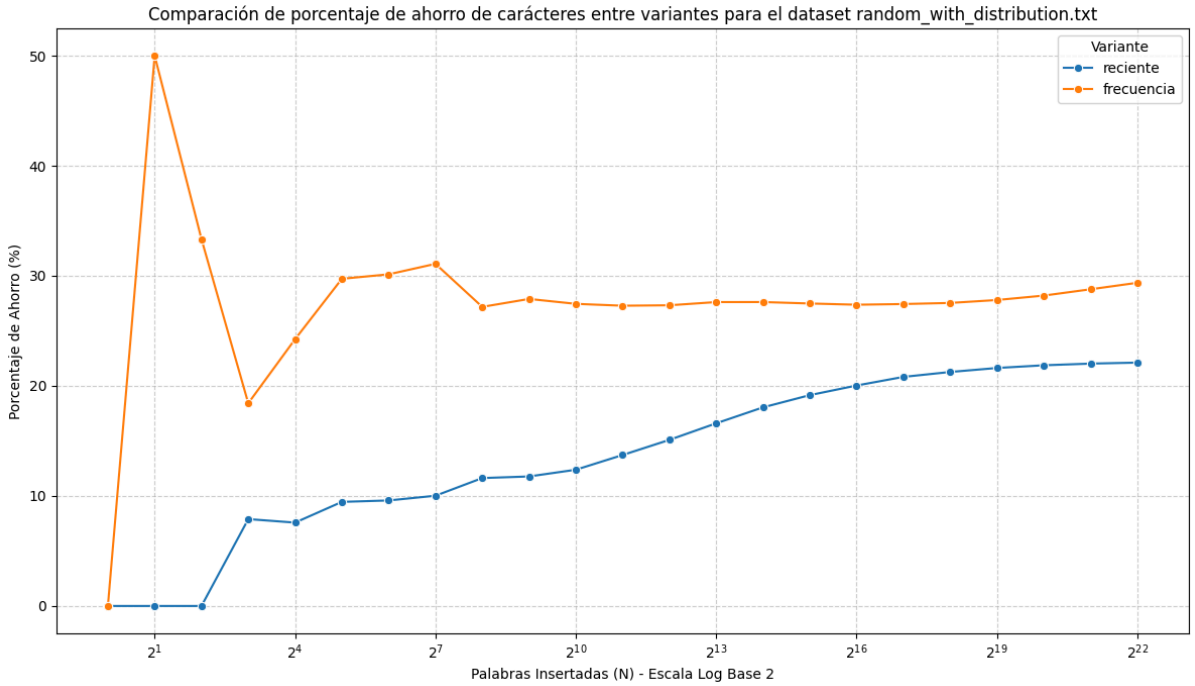
Dataset	Frecuencia (% de ahorro total)	Reciente (% de ahorro total)	Diferencia (%)	Mejor variante
wikipedia	31.313001	32.562015	1.249014	Reciente
random	21.980064	22.194733	0.214669	Reciente
random with_distribution	29.368354	22.114510	7.253844	Frecuencia

Gracias a estos datos generales podemos ver que al realizar la simulación completa de autocompletado la variante Reciente tiene un mejor porcentaje de ahorro en 2 de 3 datasets pero por una diferencia de porcentaje no muy grande menor a 2%. Mientras que en el dataset de random_with_distribution la variante de Frecuencia logra un mejor porcentaje de ahorro con una diferencia mas notoria de mas de 7%

También, podemos graficar los porcentajes de ahorro de las variantes en distintos puntos de la simulación dando como resultado unos graficos que nos dan mas detalle sobre el comportamiento de cada una de las variantes según la cantidad de palabras simuladas.







Viendo los gráficos completos sobre la variación del porcentaje de ahorro podemos ver de manera mas clara desde que punto cada variante de Trie se toma la ventaja en cada dataset.

En particular, vemos como en el dataset de Wikipedia no es hasta la inserción de 2^{10} palabras que la variante Reciente tiene mejor porcentaje de ahorro que la de Frecuencia. Es más se puede notar como no es hasta después de las 2^{12} palabras insertadas que la variante de Reciente logra una ventaja mas notoria.

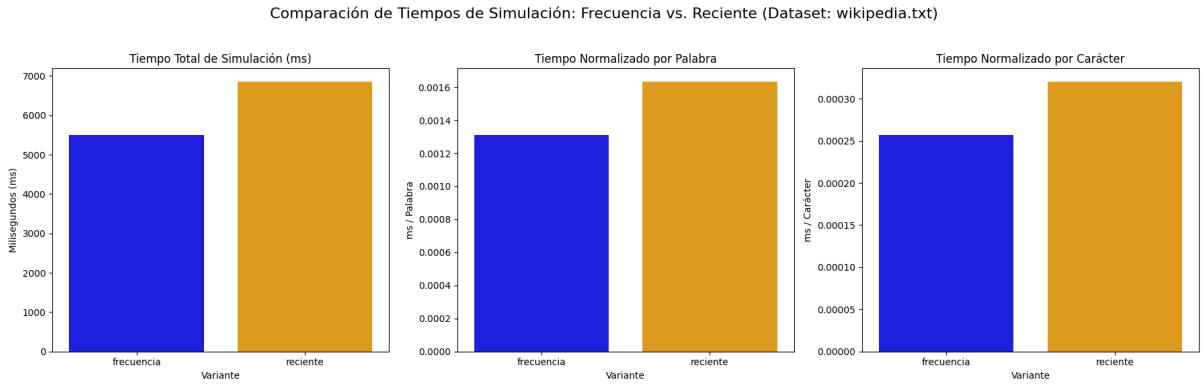
Para el dataset Random notamos como ambas variantes son prácticamente iguales con diferencias tan mínimas que son casi indistinguibles en el grafico.

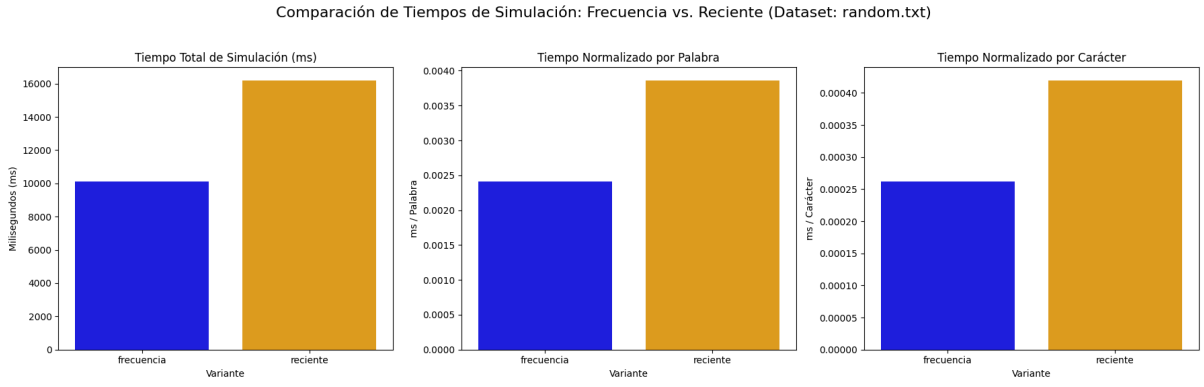
Por otro lado, para el dataset Random_with_distribution se puede observar una clara ventaja por parte de la variante de Frecuencia a lo largo de toda la simulación.

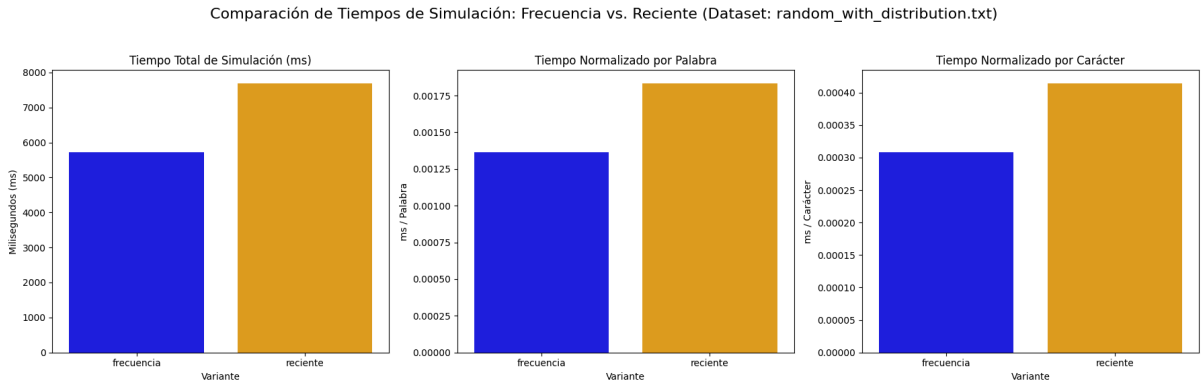
3.3.2. Tiempos de simulación

A lo largo de la simulación de autocompletado, ademas tomamos medidas de tiempo para poder saber el tiempo promedio de cada variante por palabra y por caracter.

Dataset	Palabras Totales	Caracteres Totales	Tiempo Total (ms) Frecuencia	Tiempo Total (ms) Reciente	ms Por Palabra Frecuencia	ms Por Palabra Reciente	ms Por Caracter Frecuenia	ms Por Caracter Reciente
wikipedia	4194304	21369648	5493	6846	0.001310	0.001632	0.000257	0.000320
random	4194304	38597959	10130	16176	0.002415	0.003857	0.000262	0.000419
random with distribution	4194304	18577970	5718	7683	0.001363	0.001832	0.000308	0.000414







Para la comparación de tiempo vemos como para todos los datasets el tiempo de la variante de Frecuencia siempre es menor a la variante Reciente. Siendo la diferencia mas grande en el dataset Random.

4. Análisis

5. Conclusión