Tarea 2 - Diccionarios de Strings

Profesor: Gonzalo Navarro
Auxiliar: Manuel Cáceres
Ayudantes: Jaime Salas - Tomás Perry

1. Introducción

Un *Diccionario de Strings* es un tipo de datos que almacena elementos, cuyas llaves son Strings sobre un alfabeto Σ de tamaño σ. En clases vimos los `Tries`, árboles de aridad $\sigma + 1$, cuyas aristas representan símbolos de Σ y los caminos de la raíz a las hojas representan a los `Strings` almacenados en el `Trie`. De este modo, si implementamos los nodos del `Trie` como arreglos de referencias a sus $\sigma + 1$ hijos, podemos responder si un `String` P se encuentra en el *Diccionario* en tiempo $\mathcal{O}(|P|)$ avanzando en el `Trie` al mismo tiempo que avanzamos en los símbolos de P. El problema de esta implementación es que si tenemos n `Strings` de largo total N en el `Trie` usamos espacio entre $n(\sigma + 1)$ y $N(\sigma + 1)$.

El objetivo de esta tarea es implementar y evaluar en la práctica alternativas de implementación de un *Diccionario de Strings*, comparando tiempos de construcción, de búsqueda, recorrido y espacio utilizados. Las alternativas que se les pide implementar son las siguientes:

- `Árbol Patricia`
- `Árbol de Búsqueda Ternario`
- `Hashing con Linear Probing`

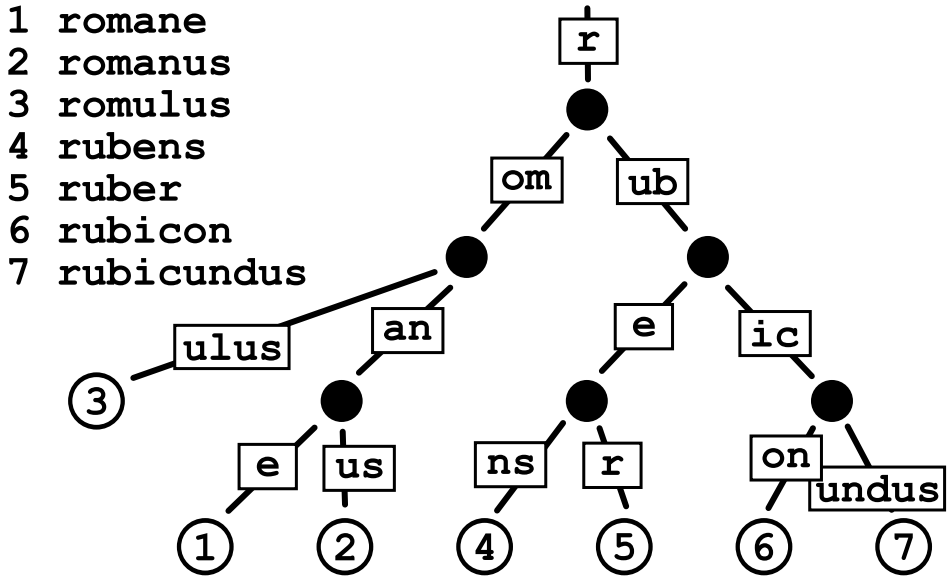
Nuevamente, se espera que se implementen las estructuras y los algoritmos correspondientes y se entregue un informe que indique claramente los siguientes puntos:

1. Las *hipótesis* escogidas antes de realizar los experimentos.
2. El *diseño experimental*, incluyendo los detalles de la implementación de los algoritmos, la generación de las instancias y las medidas de rendimiento utilizadas.
3. La *presentación de los resultados* en forma de una descripción textual, tablas y/o gráficos.
4. El *análisis e interpretación* de los resultados.

2. Las Estructuras

2.1. Árbol Patricia

Un `Árbol Patricia` [1] (también llamados Compressed Radix Tree, Compressed Prefix Tree, Compressed Trie, entre otros) es un `Trie` en el que se han comprimido las ramas unarias, reemplazándolas por arcos. En otras palabras, se reemplazan caminos unarios de la forma $N_1 \xrightarrow{c_1} \ldots \xrightarrow{c_k} N_{k+1}$ por $N_1 \xrightarrow{c_1 \ldots c_k} N_{k+1}$. Gráficamente, un `Árbol Patricia` tiene la siguiente forma:



Un `Árbol Patricia` tiene una hoja por cada palabra que almacena. Adicionalmente, un árbol de este tipo con n hijos tiene a lo sumo n nodos internos, por lo que ocupa espacio $\mathcal{O}(n)$.

Búsqueda

Para buscar una palabra $P[1, m]$ en un `Árbol Patricia`, se navega en éste utilizando las etiquetas de los nodos, comparando las subcadenas de la palabra con éstas. Si se llega a una hoja a la vez que se termina de comparar la palabra que se busca, se ha encontrado el patrón. En caso contrario, no. De esta forma, la búsqueda toma tiempo $\mathcal{O}(m)$

[1] Existen diferentes formas de ver los `Árboles Patricia`. En algunas, la palabra completa se encuentra almacenada en las hojas; en otras, los nodos intermedios almacenan subcadenas. Otras variantes utilizan etiquetas numéricas en vez de subcadenas. Estos cambios alteran ligeramente los algoritmos de inserción y búsqueda. Si decidiera utilizar una variante diferente para sus implementaciones, explicítela en el informe, detallando las diferencias con las versiones aquí expuestas y justificando su decisión.

Inserción

Para la inserción de una palabra P , se busca en el árbol: la acción que se toma depende de la razón por la que no se ha encontrado:

- Si fue porque se llegó a un nodo que no tiene un hijo que comience con la siguiente letra, se busca una hoja del nodo en el que no se pudo bajar y se reinserta P desde esa hoja.
- Si fue porque se acabó el patrón en un nodo antes de llegar a una hoja, en medio de una arista, se busca una hoja cualquiera del hijo de esa arista y se reinserta P desde ésta.
- Si fue porque se llegó a una hoja, se reinserta P desde esta hoja.

Reinserción desde una hoja

Esta subrutina se utiliza en ciertos casos de inserción, como ya se mencionó. Se tiene una palabra P que se desea insertar en la estructura, y una hoja a la que se llegó, que representa una palabra P'

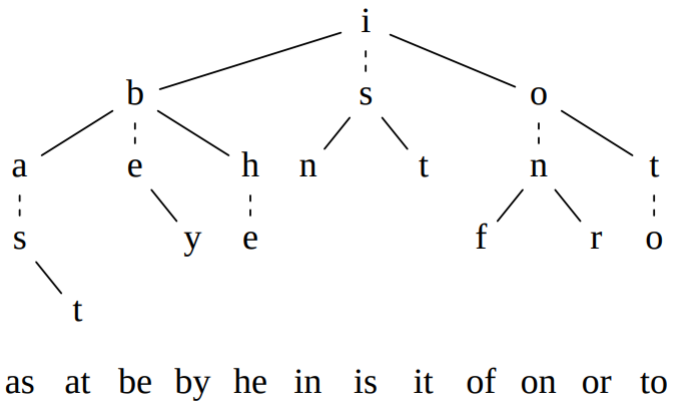
- Se compara P con P' , encontrándose p, el prefijo máximo común.
- Se entra al árbol desde la raíz, buscándose p.
  - Si p se encontró a la mitad de una arista, se corta la arista con un nodo de dos hijos: la hoja P y el hijo original de la arista.
  - Si p se encontró en un nodo, se agrega P como nuevo hijo de ese nodo.

2.2. Árbol de Búsqueda Ternario

Otra alternativa a los `Trie` son los `Árboles de Búsqueda Ternarios`(`ABT`) que son una mezcla entre un `Trie` y un `Árbol de Búsqueda`. Los `ABT` consisten en nodos etiquetados con símbolos en Σ y tres hijos. La idea es similar a la de los `Trie`; a medida que se avanza en el árbol (en profundidad) se van identificando los símbolos del `String` en el *Diccionario*, sin embargo, la identificación de un símbolo no es tan directa como lo es en un `Trie`.

Un nodo en un `ABT` etiquetado con el símbolo s tiene como hijo del centro a un `ABT` que representa los `Strings` que comienzan con el símbolo s. Por otro lado, el hijo izquierdo es un `ABT` que contiene los `Strings` que empiezan con un símbolo *menor* a s.Y de manera análoga el hijo derecho tiene un `ABT` que contiene los `Strings` que empiezan con un símbolo *mayor* a s.

Gráficamente, un `Árbol de Búsqueda Ternario` con palabras de 2 símbolos tiene la siguiente forma:



Se muestra que en *promedio* los `ABT` ocupan espacio entre $3n$ y $3N$.

Búsqueda

Al igual que en un `Trie`, para buscar un `String` $P[1, m]$, se avanza símbolo por símbolo en P. Si se está analizando $P[i]$, al entrar en un nodo:

- Si este tiene como etiqueta $P[i]$, se desciende por su hijo del centro buscando ahora $P[i+1]$.
- Si la etiqueta es menor a $P[i]$, este se busca en el hijo derecho.
- Si es mayor, se busca en el hijo izquierdo.

Si en algún momento se llega a un árbol vacío antes de procesar completamente P, entonces P no es parte del *Diccionario*. Se muestra que en *promedio* esto toma $\mathcal{O}(m + \ln n)$.

Inserción

Para insertar en un `ABT`, primero se hace una búsqueda de P. Sea $P[i, m]$ el sufijo del `String` que faltó por procesar y r la referencia al nodo *vacío* donde terminó la búsqueda de P. Lo que se hace entonces es insertar $P[i, m]$ en r como si fuera un `Trie`, es decir, creando nodos unidos por su referencia central que tienen como etiquetas los símbolos de $P[i, m]$.

2.3. Hashing con Linear Probing

Sea $h : \Sigma^* \to \mathbb{Z}$ una función de hash para `Strings` y $T[0, k-1]$, con $n \leq k$, un arreglo de `Strings` llamado *tabla de hash*. Otra forma de implementar un *Diccionario de Strings* consiste en guardar los `Strings` en la *tabla de hash*, en las posiciones dadas según la función de hash. En caso que exista una *colisión*[2] se prueba en la siguiente posición de T sucesivamente. A este método de resolución de colisiones se le conoce como *Linear Probing*.

Búsqueda

Para buscar un `String` $P[1, m]$, primero calculamos su *función de hash* $f \leftarrow h(P)$ y luego su posición en T, $i \leftarrow f \ mod \ k$. Con esto hacemos una búsqueda secuencial en T desde $T[i]$(volvemos al principio si llegamos al final de la tabla) comparando con P hasta que lo encontramos. En caso que no lo encontremos (lleguemos a una entrada vacía en la tabla) P no es parte del diccionario. Considerando suposiciones razonables sobre la función de hash utilizada y la ocupación de la tabla esta implementación toma tiempo $\mathcal{O}(m)$.

Inserción

Para insertar un String $P[1, m]$ calculamos nuevamente su posición i en T y desde $T[i]$ buscamos secuencialmente(volvemos al principio si llegamos al final de la tabla) el primer lugar vacío $T[j]$ donde ubicamos a P. Considerando suposiciones razonables sobre la función de hash utilizada y la ocupación de la tabla esta implementación toma tiempo $\mathcal{O}(1)$.

El espacio ocupado por esta implementación es $\mathcal{O}(N + k)$.

3. Aplicaciones

3.1. Búsqueda en Texto

Para implementar *Búsqueda en Texto* con estas estructuras:

- Separamos el `Texto` en palabras.
- Utilizamos estas palabras como el conjunto de llaves del *Diccionario*.
- Los valores asociados a las llaves corresponden a las posiciones en donde se encuentra la palabra en el `Texto`.

De este modo cuando busquemos si una palabra está en la estructura podemos determinar eficientemente si ésta está en el `Texto`, cuántas veces aparece, y cuáles son sus ubicaciones en él.

[2]Dos string que dan el mismo valor de hash.

3.2. Simlitud entre Textos

Se les pide también medir la *similitud* entre dos `Textos`. Para esto consideraremos un índice de similitud. Si T_1 y T_2 son 2 textos(a los que les queremos calcular su similitud), T_1T_2 será su concatenación y P el conjunto de palabras presentes en esta concatenación. Definimos entonces la *similitud* entre T_1 y T_2 como:

$$similitud(T_1, T_2) = 1 - \frac{\sum_{s \in P} |count(s, T_1) - count(s, T_2)|}{count(T_1T_2)}$$

donde *count* retorna el número de palabras s presentes en el `Texto` (si s no es especificado, retorna el número de palabras del `Texto`).

Para calcular la *similitud* entre T_1 y T_2:

- Separamos ambos `Textos` en palabras.
- Utilizamos estas palabras como el conjunto de llaves del *Diccionario*.
- Los valores asociados a las llaves corresponden a un par que representa la frecuencia de aparición de la palabra en cada uno de los `Textos`, T_1 y T_2.

Para calcular la *similitud* hacemos un recorrido en los valores del *Diccionario*.

4. Datos y Experimentos

Utilice, al menos, textos de lenguaje natural. Preprocese los textos que usará: elimine saltos de línea, puntuación, lleve todo a minúsculas y otras operaciones que estime convenientes. Luego de este preprocesamiento, asegúrese de que sus textos tengan largos (al menos aproximados) de $n = 2^i$ palabras, con $i \in \{10, 11, \ldots, 20\}$, asegúrese también de tener al menos 2 textos diferentes de cada largo para realizar las pruebas de similitud.
Para cada texto, construya cada una de las estructuras midiendo sus tiempos de construcción correspondientes, también *estime* el espacio ocupado por cada una de ellas. Para la *la tabla de hash* asegúrese que el porcentaje de llenado de esta nunca supere el 40 %, además utilice una función de hash apropiada para `Strings`, no olvide documentar este dato también. Para los Árboles agregue cada palabra junto a un símbolo $\$ \notin \Sigma$ (y menor lexicográficamente a cualquier otro símbolo) concatenado al final, de este modo será más fácil identificar palabras que son prefijos de otras en el *Diccionario*.

Escoja $n/10$ palabras de forma aleatoria del texto y encuentre todas las ocurrencias de éstas, registrando los tiempos de búsqueda en función del largo m del patrón para cada una de las estructuras. Escoja también palabras que no se encuentren en el texto y registre los tiempos de "search miss" dependiendo del largo m.

Finalmente, escoja pares de textos de tamaño similar y calcule su similitud. Tome los tiempos de inserción de las palabras y los tiempos de "recorrido" de la estructura (parte final del calculo de similitud). Repita estos procesos (construcción, búsquedas y recorridos) para obtener promedios confiables para los tiempos registrados.

Note que tiene libertad para escoger los textos que utilizará: de esta forma, puede escogerlos para ayudarse a poner a prueba su hipótesis, si fuera necesario. Otro grado de libertad es el tamaño del alfabeto con el que trabajará (por ejemplo, puede comparar lo que sucede al utilizar lenguaje natural versus binario).

5. Entrega de la Tarea

- La tarea puede realizarse en grupos de a lo más 3 personas.
- Para la implementación puede utilizar C, C++ o Java. Para el informe se recomienda utilizar LaTeX.
- Siga buenas prácticas (*good coding practices*) en sus implementaciones.
- Escriba un informe claro y conciso. Las ponderaciones del informe y la implementación en su nota final son las mismas.
- Tenga en cuenta las sugerencias realizadas en las primeras clases sobre la forma de realizar y presentar experimentos.
- La entrega será a través de U-Cursos y deberá incluir el informe junto con el código fuente de la implementación (y todas las indicaciones necesarias para su ejecución).
- Se permiten atrasos con un descuento de 0.5 puntos por día.

6. Links

- En http://www.gutenberg.org puede encontrar una gran cantidad de documentos.
- Otra fuente de datos: http://pizzachili.dcc.uchile.cl/texts/nlang/ de lenguaje natural.
- En la Introducción de esta publicación se puede encontrar una pequeña explicación de *Linear Probing*: `algo.inria.fr/flajolet/Publications/RR3265.ps.gz`
- Publicación de Árbol de Búsqueda Ternario: `http://www.cs.princeton.edu/~rs/strings/paper.pdf`
- En la sección 7.4 de `http://cglab.ca/~morin/teaching/5408/notes/strings.pdf` puede encontrar una versión optimizada en espacio de los `Árboles Patricia`.