Proyecto 1: Texto Predictivo

Estructuras de Datos, Primer Semestre 2020

Prof. Diego Seco

Ayudantes: Catalina Pezo y Alexis Espinoza

1. Descripción del Problema

En este proyecto se pide implementar un sistema de predicción de texto simple para el español o inglés. El predictor incorporará una base de datos de palabras, y será construido a partir de una lista de palabras, organizando la información de tal forma que se pueda hacer una búsqueda eficiente. Cada palabra tendrá además asociado un número entero representando la frecuencia con que se ocupó dicha palabra.

Las cadenas de caracteres a ser predichas (prefijos de palabras) se leen, una a una, de un archivo de entrada (de texto) y son entregadas al predictor para su procesamiento. El predictor buscará entre todas las palabras que comiencen con dicha cadena y retornará las k más frecuentes. Si hay menos de k resultados, eliminará el último caracter de la cadena solicitada y realizará el mismo proceso con la cadena resultante. Este paso se puede repetir varias veces hasta completar los k resultados. Por ejemplo, si la cadena a buscar es "predicto" y $k = 3$, y la única palabra que contiene el sistema que comience con "predicto" es "predictor", entonces se incorporará "predictor" a los resultados y se repetirá el proceso con $k = 2$ y la cadena de búsqueda "predict".

En caso de que haya empates en la frecuencia, el criterio de desempate será orden alfabético. Para todas las palabras que se incluyen en la lista de resultados, se incrementará su frecuencia en 1.

2. Implementación

El programa debe ser llamado por línea de comandos recibiendo como parámetros: el archivo que contiene las palabras del diccionario, el archivo con las palabras a buscar y el nombre del archivo que contendrá los resultados. Una llamada al programa debe ser de la forma:

```
predictor <archivo_diccionario> <cadenas_a_buscar> <resultados> <k>
```

El programa debe verificar la existencia de los dos primeros archivos de entrada e imprimir un error si alguno no existe. El contenido y formato de los archivo de entrada `archivo_diccionario` y `cadenas_a_buscar`, y salida `resultados` se encuentra detallado más abajo.

Existen diversas formas de implementar un predictor de texto. En este proyecto la implementación debe ajustarse a las siguientes restricciones:

- El diccionario de palabras debe almacenarse en un árbol especial denominado *trie* (ver Anexo A). Esta sencilla pero potente estructura de datos no se suele ver en el curso básico de Estructuras de Datos pero, dada su simplicidad y semejanza con cualquier otra implementación de árboles, la descripción incluida en el anexo debería ser suficiente para su implementación.

- El Predictor debe ser implementado como una clase. Contendrá una instancia de Trie, pero no será únicamente un trie.
- Debe existir una clase "Controladora"que se preocupe de procesar los archivos de entrada y de crear el archivo de salida. Esta clase contendrá una instancia de la clase Predictor (y probablemente otras variables más).
- En la implementación se deben respetar los principios de buen diseño, abstracción y encapsulamiento. Por ejemplo, debe definir una interfaz para el Predictor, el árbol no debe asumir responsabilidades que no le pertenezcan como la lectura de cadenas a procesar o la construcción de la lista de resultados, la representación debe ser privada y las modificaciones y consultas se deben hacer sólo a través de métodos.

3. Archivos

Los archivos de entrada y salida del programa son los siguientes:

(1) `archivo_diccionario`**:** archivo de entrada que contiene una palabra por línea. Puede contener comentarios partiendo con el símbolo %. Pueden encontrar ejemplos de diccionarios aquí: `http://www.winedt.org/dict.html`. Asuma que todas las palabras están con minúsculas. Por simplicidad, para cada palabra se inicializará su frecuencia con un número aleatorio en el intervalo [0-1000].

(2) `cadenas_a_buscar`**:** archivo que contiene las cadenas a buscar cada una en una línea. Asuma que están con minúsculas.

(3) `resultados`**:** este archivo de salida debe contener las k predicciones para cada cadena. Debe contener tantas líneas como cadenas buscadas y, en cada una, `<cadena_buscada>: <predicción 1> ... <predicción k>`. Es importante mantener este formato para poder automatizar parcialmente la corrección de la tarea.

4. Evaluación

El proyecto se realizará en parejas. Subir en Canvas lo siguiente:

1. Un informe en pdf que:
   a) Incluya portada, descripción de la tarea, descripción de la solución propuesta (puede emplear diagramas de clase para explicar la solución) y detalles de implementación. Además, debe incluir una evaluación preliminar de tiempo y espacio empleados por el trie. Para eso se trabajará con diccionarios de 100, 500, 1000, 5000, 10000, 50000 y 100000 palabras. Se incluirá un gráfico donde el eje X se corresponderá con número de palabras y el eje Y con memoria usada por la estructura. También se incluirá un gráfico donde el eje X se corresponderá con número de palabras y el eje Y con tiempo consumido para la construcción del trie con dicho conjunto. Si alguno de los experimentos toma más de 30 minutos, u ocupa toda la memoria del computador, se indicará en el informe con un texto como "No se pudieron procesar conjuntos de datos de más de X elementos debido a que el experimento llevaba más de 30 minutos (alt. no se disponía de suficiente memoria)."
   b) Sea claro y esté bien escrito. Un informe difícil de entender es un informe que será mal evaluado aunque todo esté bien implementado. La persona que revise el documento debe poder entender su solución sólo mirando el informe.
2. Un archivo comprimido con todos los ficheros fuente implementados para solucionar la tarea. El informe debe hacer referencia a ellos y explicar en qué consiste cada uno.

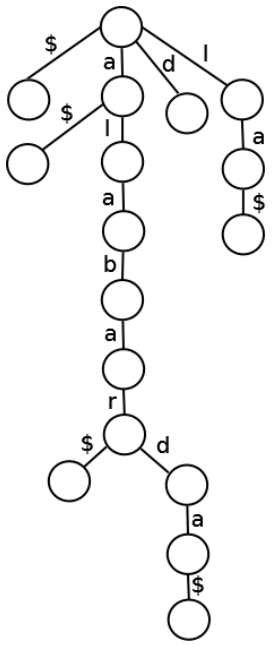
Fecha de Entrega: viernes 29 de Mayo 11:59PM

A. Trie

Un trie[1] permite almacenar un conjunto de strings $T_1, ..., T_k$ soportando búsquedas (predecesor/sucesor) en tiempo proporcional al largo del patrón buscado: $O(m)$.

Estructura: Árbol donde cada rama hija está etiquetada con un símbolo del alfabeto $\sum$.
Conceptualmente, todos los strings deben estar terminados por un símbolo especial cuyo valor de codificación es menor a todos los otros símbolos de $\sum$. En este documento se usará el caracter $ para ejemplificar.
Todos los recorridos desde el nodo raíz hasta los nodos hojas corresponden a un string almacenado.

Ejemplo: {'alabar$', 'a$', 'la$', 'alabarda$' }

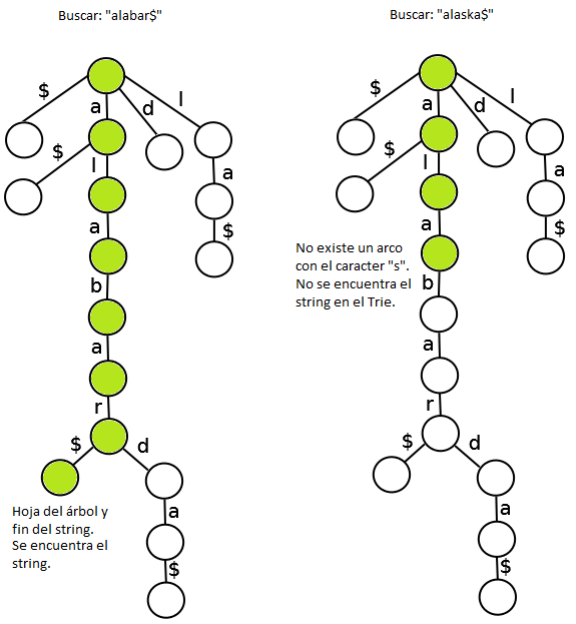


Para buscar un string en un Trie se empieza en la raíz del árbol, buscando un subárbol hijo cuyo arco que los conecta contenga el primer caracter del string. Si éste no existe, entonces el string buscado no está almacenado en el Trie. De lo contrario se procede de forma recursiva, esta vez con la raíz del subárbol y el segundo caracter del string (y así sucesivamente).

La búsqueda termina cuando se llega al último caracter del string. Si el último nodo visitado es una hoja, entonces el string se encuentra en el Trie, de lo contrario el string no se encuentra.
En la práctica, se puede emplear el símbolo especial $ (debe ser alguno que no esté en las palabras introducidas), o bien los nodos pueden contener un campo booleano que indiquen si el nodo corresponde al término de un string. Ambas son implementaciones válidas.

Ejemplo: Buscar "alabar$" y "alaska$" en el Trie del ejemplo anterior.

[1] El nombre Trie procede de information reTRIEval.



¿Qué nos proporciona un recorrido in-order de las hojas?

- Strings en orden lexicográfico.

¿Complejidad de búsqueda de un string de largo m?

- $O(m \times access\_child)$
- Por tanto, depende de cómo se soporte la operación *access_child* en la implementación concreta. Las más populares son:

	Espacio	Tiempo
List	O(T)	O(mσ)
Vector	O(T$\sum$)	O(m)
BST	O(T)	O(m log$\sum$)
Hash	O(T)	O(m) w.h.p.

Donde T es el número de nodos en el árbol, $\sum$ es el número de caracteres en el alfabeto.

Otras implementaciones (para soportar predecesor/sucesor):

	Espacio	Tiempo
vEB/y-fast	O(T)	O(mloglog$\sum$)
Suffix trays	O(T)	O(m+log$\sum$)

En este proyecto se puede optar por una implementación basada en Vector (arreglo) o por una basada en List. En cualquier caso, se debe argumentar bajo qué circunstancias (descripción del dominio) esa sería la mejor opción.