UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Práctica Final: El juego de las letras
Práctica 4: Estructura de árbol para el diccionario y Solver

(Práctica puntuable)

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Estructuras de Datos

Grado en Ingeniería Informática
Doble Grado en Ingeniería Informática y Matemáticas
Doble Grado en Ingeniería Informática y ADE

1.- Introducción

1.1 Juego de las letras

En esta práctica retomaremos el juego de las letras, manejando una estructura de datos alternativa para tratar de dar una solución más eficiente al mismo. Recordemos en qué consiste una partida del juego de las letras:

El objetivo es formar la mejor palabra posible (dependiendo de uno de los dos criterios que explicamos a continuación) a partir de un conjunto de letras extraídas al azar de una bolsa. Por ejemplo, dadas las siguientes letras:

O D Y R M E T

Una buena solución posible sería METRO. El número de letras que se juegan en cada partida se decide de antemano, y las letras disponibles pueden repetirse. Existen dos modalidades de juego:

- Juego a longitud: En este modo de juego se tiene en cuenta sólo la longitud de las palabras, y gana la palabra más larga encontrada.
- Juego a puntos: En este modo de juego a cada letra se le asigna una puntuación, y la puntuación de la palabra será igual a la suma de las puntuaciones de las letras que la componen.

En esta práctica modificaremos la implementación del TDA Dictionary de la práctica anterior utilizando como contenedor subyacente un árbol LCRS (*Left-Child Right-Sibling*). De esta forma, podremos almacenar nuestro diccionario con menos espacio y crear un solver más eficiente. Concretamente:

1. Necesitaremos almacenar menos información para representar el diccionario completo (lo podremos comprobar contando el número de letras almacenadas en total).
2. Podremos recuperar más rápidamente las palabras que se pueden formar dado un vector de letras, ya que la estructura de árbol en la que almacenaremos el diccionario nos permitirá explorar las posibles soluciones de una forma bastante eficiente.

1.2 Información necesaria para una partida de las letras: Archivos de entrada

Información sobre las letras

Como hemos dicho anteriormente, uno de los modos de juego a los que podemos jugar asigna una puntuación a cada letra, y la puntuación de la palabra será la suma de las puntuaciones de cada una de sus letras. Por tanto, necesitamos recoger la información de la puntuación para cada letra de algún sitio.

Además, hemos dicho que podríamos tener cada letra repetida un número de veces. No obstante, esto puede llevar a problemas en algunas partidas. Si todas las letras pudieran repetirse un número indeterminado de veces, podría ocurrir que en alguna partida sólo tuviésemos la letra Z muchas veces, lo que dificultaría en gran medida formar una palabra. Además, parece lógico pensar que, si las letras que más se repiten son letras que aparecen mucho en el diccionario, las palabras que se podrán formar serán más largas, haciendo el juego más interesante para los participantes. Por esto, la forma que tendremos de seleccionar las letras de cada partida será considerando un número de repeticiones de cada letra, formando una “bolsa” con todas ellas, y sacando al azar con

probabilidad uniforme elementos de esa bolsa. Así, las letras con más repeticiones tendrán mayor probabilidad de salir, mientras que las que tengan pocas repeticiones saldrán menos a menudo (y no se repetirán en la misma partida si no hay más de una copia, ya que haremos extracciones sin reemplazamiento).

Por estos dos motivos, tendremos que guardar la siguiente información para cada letra del abecedario:

- Su puntuación.
- El número de repeticiones disponible.

Esta información la leeremos de un fichero como el que se muestra a continuación:

```
Letra  Cantidad Puntos
A      12       1
B      2        3
C      5        3
D      5        2
E      12       1
F      1        4
G      2        2
H      2        4
I      6        1
J      1        8
L      1        1
M      2        3
N      5        1
O      9        1
P      2        3
Q      1        5
R      6        1
S      6        1
T      4        1
U      5        1
V      1        4
X      1        8
Y      1        4
Z      1        10
```

Fichero 1: letras.txt

Crearemos una estructura de datos adecuada para almacenar esta información durante una partida, y la aprovecharemos para calcular las puntuaciones de nuestras palabras. Esta estructura de datos se denominará Conjunto de Letras (TDA LettersSet).

Además, construiremos un contenedor específico para extraer las letras de forma aleatoria para una partida en función a las repeticiones indicadas por cada letra del archivo anterior. Dicho contenedor será capaz de leer un Conjunto de Letras con las repeticiones de cada una de las letras del juego, y crear la bolsa de la que haremos las extracciones aleatorias. Este TDA se denominará Bolsa de Letras (TDA LettersBag).

Para estas dos estructuras, mantendremos la implementación que habíamos propuesto en las prácticas anteriores (en particular, la práctica 3). Por tanto, mantendremos sin modificar los TDAs:

- LettersSet (implementado como un `map<char, LetterInfo>`).

- Bag (implementado como una clase template que nos permite extraer elementos aleatorios).
- LettersBag (implementado como una `Bag<char>`).

Información sobre las palabras - Diccionario

Finalmente, dado que vamos a jugar a un juego que trabaja sobre palabras, es importante establecer qué palabras estarán permitidas en una partida. Una “palabra” formada con las letras del ejemplo anterior puede ser METROYD, pero claramente no podríamos aceptar esta palabra en una partida, ya que no pertenece a nuestro idioma.

Por tanto, necesitaremos trabajar con un listado de palabras permitidas en nuestro juego, que funcionará a modo de diccionario. Sólamente aceptaremos como válidas aquellas palabras que pertenezcan a nuestro diccionario. La información de las palabras la recopilaremos de un archivo como el siguiente:

```
a
aaronita
aaronico
aba
ababa
ababillarse
ababol
abacal
abacalero
...
```

Fichero 2: diccionario.txt

La principal modificación que introduciremos en esta práctica a las estructuras de datos que implementamos en la práctica anterior tiene que ver con la clase Diccionario. En la siguiente sección, se comenta en detalle dicha modificación, que estará centrada en reducir la cantidad de información almacenada a la hora de guardar el diccionario.

2.- Una nueva forma de guardar el diccionario - estructura de árbol

Lo primero que tenemos que observar, para dar sentido a la modificación propuesta, es la cantidad de información redundante que estamos almacenando cuando utilizamos un conjunto para guardar nuestro diccionario. Aunque no tenemos palabras repetidas en nuestro set, no estamos aprovechando una propiedad que cumplen las palabras que estamos almacenando: muchas de ellas comparten un prefijo común, más o menos largo, que en algunos casos llega a ser la palabra completa. Tomando como referencia las palabras que se muestran en el recuadro anterior, por ejemplo, tenemos que las palabras aaronita y aaronico comparten un prefijo de 6 letras, y sólo se diferencian en las últimas 2. Igualmente, aba está completamente contenida en ababa, y ésta a su vez está incluida casi por completo en ababillarse. Si encontramos una estructura en la que podamos almacenar las palabras de forma que compartan esos prefijos, podremos ahorrar mucho espacio eliminando toda la información redundante. Para ello, vamos a hacer uso de una estructura de árbol.

2.1 Módulo tree - implementación de un árbol general en formato LCRS

El módulo tree, que se proporciona completamente implementado, representa un árbol general en el que cada nodo puede tener un número de hijos arbitrario. Para ello, cada nodo almacena la siguiente información:

- La etiqueta del nodo, es decir, la información que dicho nodo almacena, y que puede ser de cualquier tipo (para ello utilizamos una template de C++).
- Un puntero al nodo hijo más a la izquierda.
- Un puntero al nodo hermano de la derecha.
- Un puntero al nodo padre.

Esta forma de guardar un árbol es lo que se conoce como la representación LCRS (Left Child Right Sibling). Permite que un nodo padre itere por todos sus hijos visitando a su hijo más a la izquierda, y recorriendo a partir de ahí los nodos hermanos de la derecha hasta encontrar un nodo nulo. Así, el árbol queda completamente determinado (podemos recorrerlo por completo) simplemente almacenando un puntero al nodo raíz.

En nuestro caso, como queremos almacenar un conjunto de palabras, almacenaremos en cada nivel del árbol un carácter de la palabra, y reconstruiremos las palabras cogiendo todos los caracteres desde el nodo raíz (que contendrá un carácter especial y no utilizaremos para formar palabras) hasta un nodo determinado del árbol, haciendo un recorrido en profundidad.

Para poder determinar qué nodos nos permiten formar una palabra de nuestro diccionario, tendremos que poder indicar si en cada nodo termina o no una palabra. Por tanto, la etiqueta de cada nodo contendrá dos valores: un carácter, que representará una determinada letra de las palabras que se formen por esa rama, y un valor booleano, que nos indicará si en ese punto termina una palabra. Esta información la guardaremos en una estructura `struct char_info`, para poder instanciar un árbol que almacene estos dos valores en sus etiquetas.

Se puede ver un ejemplo de árbol almacenando un conjunto de palabras en la figura 1. En concreto, en dicho ejemplo se almacenan las palabras:

- a
- ama
- amar
- amo
- amor
- cal
- col
- coz

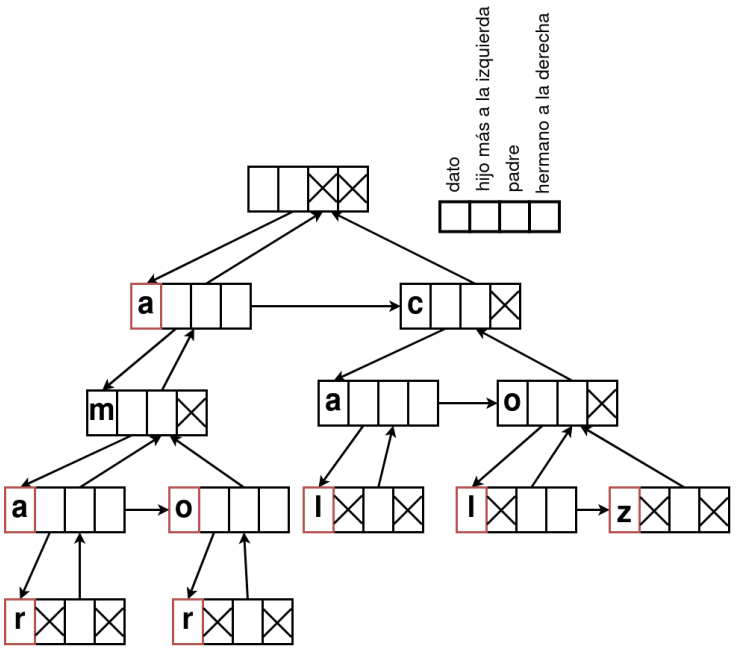


Figura 1: *Ejemplo de árbol representando un conjunto de palabras. Los nodos de color rojo indican el final de una palabra. Las palabras pueden reconstruirse viajando de la raíz a un nodo en rojo.*

Al almacenar la información en este tipo de estructura, obtendremos dos ventajas:

- La cantidad de información almacenada se reduce, ya que para las palabras que compartan prefijo sólo almacenaremos una copia de cada elemento que compartan. En el ejemplo anterior, tenemos que almacenar 24 caracteres entre todas las palabras de nuestra lista, mientras que en el árbol sólo hay 12 nodos (13 si contamos la raíz, que no almacena ningún carácter).
- Buscar en el diccionario si hay una determinada palabra es rápido, ya que sólo tendremos que buscar, cuando nos encontremos en un determinado nodo, si entre sus hijos se encuentra la siguiente letra de nuestra palabra. Si en un determinado nivel no está la letra que nos toca buscar, la palabra no podrá estar en el diccionario, y si conseguimos encontrar todas las letras en orden, la palabra pertenecerá al diccionario si el último carácter está marcado como fin de palabra (es decir, si el booleano del struct que almacena el nodo es true).

La implementación completa del árbol se puede encontrar en los archivos `tree.h`, `tree.tpp` y `node.tpp`, que se encuentran dentro de la carpeta include. En principio, las clases plantilla deben declararse e implementarse en el mismo archivo. El uso de archivos `.tpp` permite separar en cierto modo la implementación de la declaración, pero todos los archivos que implementen una clase template deben estar en una carpeta de cabeceras.

Iterador para la clase árbol

La estructura de árbol que hemos implementado es muy útil para almacenar la información, pero resulta complicado recuperar cómodamente la información contenida. Este es el motivo por el que se abstrae el recorrido de las estructuras de datos utilizando iteradores. Un iterador es una clase que nos permite recorrer una estructura de datos determinada en un orden concreto, definido de antemano. En nuestro caso concreto, recorreremos la estructura de árbol en preorden, ya que nos permitirá a posteriori recuperar las palabras de nuestro diccionario en orden alfabético. Para un árbol en formato LCRS, el recorrido en preorden funciona del siguiente modo. Dado un nodo de un árbol en formato LCRS:

- Visitamos el nodo actual.
- Visitamos en preorden su hijo más a la izquierda.
- Visitamos en preorden su hermano a la derecha.

Nuestro iterador tiene que soportar la funcionalidad suficiente para recorrer en árbol en preorden. En concreto, nuestro iterador almacenará en su parte privada el nodo al que estamos apuntando actualmente, y necesitaremos implementar los siguientes métodos para poder recorrer el árbol por completo:

- Constructor por defecto: Creará un iterador que apunte a un nodo nulo.
- Constructor con parámetros, que reciba un nodo: Creará un iterador cuyo nodo actual sea el argumento que hemos pasado.
- Operadores de igualdad y distinto: Estos operadores compararán si el iterador está apuntando o no a un mismo nodo, es decir, si sus nodos internos son iguales.
- Operador *: Devolverá el contenido del nodo al que estamos apuntando en este momento.
- Operador ++: Nos permitirá avanzar al siguiente nodo. El siguiente nodo se define como:
    - Nuestro hijo a la izquierda si no es nulo.
    - Si nuestro hijo es nulo, nuestro hermano a la derecha si no es nulo.
    - Si nuestro hermano también es nulo, el siguiente nodo a visitar es el hermano a la derecha de nuestro padre (a nuestro padre lo visitamos antes de bajar a sus hijos, por tanto no tenemos que volver a él).

- Si el hermano de nuestro padre también es nulo, seguiremos buscando en el hermano derecho de nuestros ancestros hasta que encontremos un nodo no nulo.
- Habremos acabado de recorrer el árbol cuando lleguemos a la raíz, es decir, cuando encontremos un nodo cuyo padre sea nulo. En ese caso, pondremos el nodo actual al nodo vacío para marcar que hemos terminado de recorrer el árbol.

Además, aunque no es información estrictamente necesaria para el funcionamiento del iterador, almacenaremos también la profundidad (o nivel) a la que nos encontramos en el árbol. Esta información nos será útil para las implementaciones que tenemos que hacer después (en concreto, para iterar por el diccionario). El control de la profundidad a la que nos encontramos es relativamente sencillo. En primer lugar, cuando se crea el iterador, se inicializa el nivel a 0. Cuando visitemos un nodo hijo, aumentamos el nivel en una unidad. Si nos movemos a un hermano a la derecha, mantenemos el nivel en el mismo valor que estaba, ya que no hemos cambiado de nivel. Si visitamos el hermano a la derecha de alguno de nuestros ancestros, decrecentamos el nivel tantas unidades como niveles hayamos ascendido.

Finalmente, son necesarias las funciones begin_preorder y end_preorder, que apuntarán al nodo raíz y a un nodo nulo, respectivamente, y que nos permitirán inicializar el iterador y aportar la condición de parada cuando terminemos de recorrer el árbol. Este iterador se aporta implementado, y puedes encontrarlo en los archivos `tree.h` y `tree.tpp`.

2.2 Clase diccionario

Una vez tenemos nuestra estructura de árbol definida e implementada, vamos a utilizarla para gestionar nuestro diccionario. Como ya hemos dicho, lo que almacenamos en cada nodo del árbol es una letra y un bool, que nos indica si en dicho nodo finaliza una palabra. Así, podremos encontrar las palabras de nuestro diccionario en orden alfabético recorriendo el árbol en preorden.

Este diccionario tendrá que tener la siguiente funcionalidad, al igual que teníamos en la práctica anterior:

- Constructor por defecto, que nos inicializa el árbol vacío.
- Constructor de copia, que copia un diccionario en otro.
- Destructor, que destruye el diccionario, liberando la memoria que hayamos reservado.
- Una función para borrar todos los nodos de nuestro diccionario, `clear()`.
- Una función que devuelva el número de palabras almacenadas en el diccionario, `size()`.
- Una función que compruebe si el diccionario está vacío (no contiene palabras), `empty()`.
- Una función para comprobar si una palabra existe en el diccionario, `exists(string word)`, que va recorriendo el árbol en profundidad buscando si la siguiente letra de la palabra que estamos buscando está entre los nodos hijos del nodo actual. Si conseguimos llegar a la última letra encontrándolas todas, y el bool del último nodo que visitemos es `true`, sabemos que nuestra palabra se encuentra en el diccionario.
- Una función para insertar una nueva palabra en nuestro diccionario, `insert(string word)`, que irá iterando en profundidad nuestro árbol e insertando cada una de las letras de la palabra (si la letra ya existe, no hacemos nada, simplemente seguimos insertando a partir de ese nodo las letras siguientes), y pondrá a `true` el booleano del último nodo que visitemos.
- Una función para eliminar palabras, `erase(string word)`, que buscará si esa palabra existe en el diccionario, y si existe, pondrá a `false` el booleano de su último nodo, para que dejemos de considerar esa palabra como parte del diccionario.
- El operador de asignación, que nos permita copiar el contenido de un árbol a otro.
- La sobrecarga de los flujos de entrada y salida, que nos permitirán cargar nuestro diccionario desde un flujo de entrada e imprimir el diccionario en un flujo de salida.

Además de estas funciones públicas, que nos dan la interfaz completa necesaria para trabajar con el Diccionario desde fuera de la clase, hay varias operaciones que son interesantes porque se realizan muchas veces en el resto del código, pero que no tienen sentido como funciones públicas, ya que son para la gestión interna de nuestra estructura de árbol. Las implementaremos, por tanto, como funciones privadas. Estas funciones son las siguientes:

- Una función que, dado un nodo del árbol, nos busque la posición de inserción de un hijo suyo de forma ordenada. Dado que estamos trabajando con una estructura de diccionario en la que nos interesa que todas las palabras estén ordenadas en orden alfabético, nos interesará introducir las letras hijas de cada nodo de forma ordenada (si se observa el gráfico del apartado anterior, los hijos de cada nodo estaban ordenados por el orden alfabético). Un ejemplo claro son los hijos de la m de la izquierda, primero aparece la a, y después aparece la o. Si quisiéramos insertar ahora la palabra americano, la e se debería insertar entre estos dos elementos. Así que, dado un nodo y una letra a insertar como hija, nos interesará devolver el último hijo de dicho nodo cuyo carácter sea menor o igual que el carácter que queremos insertar. Así, podremos insertar la nueva letra como un hermano a la derecha de dicho nodo. La función que devuelve dicha posición es la función `findLowerBoundChildNode(char character, node current)`
- Tras esta operación, como vamos a insertar muchos caracteres, también nos interesará separar esta funcionalidad en una función nueva (no es la misma operación buscar un nodo que insertarlo). Dicha función, `insertCharacter(char character, node current)`, intentará insertar el carácter como un nuevo nodo hijo del nodo current, y nos devolverá el nodo que se ha creado. Si entre los hijos de current ya existía un nodo con la letra que estamos intentando insertar, simplemente nos devolverá ese nodo, para que podamos seguir insertando por ahí, pero no tendría que insertar nada porque ya tenemos el hijo que queríamos dentro de nuestro árbol.

Todas estas funciones se os entregan ya implementadas. Con esta funcionalidad, ya podemos sustituir el diccionario implementado en la práctica 4 por este nuevo, sin necesidad de cambiar ninguna de las funciones que habíamos implementado en el resto de las clases de dicha práctica.

Ejercicio 1 - Iterador del diccionario

La estructura que hemos definido es muy eficiente a la hora de almacenar la información, pero es poco útil para recuperar las palabras que hay almacenadas en nuestro diccionario. Por tanto, para facilitar la iteración sobre dichas palabras, vamos a implementar un iterador que nos permita visitarlas por orden alfabético. Para ello, haremos uso del iterador del árbol en preorden que se ofrece implementado. En concreto, el iterador del diccionario irá recorriendo cada uno de los nodos de nuestro árbol, y parándose en cada uno de los nodos que esté marcado como final de palabra válida. Este iterador almacenará en su interior un iterador del árbol y una string que representa la palabra actual que estamos visitando, y tendrá los siguientes métodos implementados:

- Constructor por defecto, que inicializará la palabra actual a la cadena vacía y el iterador interno al iterador por defecto del árbol.
- Constructor por parámetros, que recibirá otro iterador del diccionario y copiará tanto la string como el iterador del árbol al nuevo iterador del diccionario.
- Operador *, que devolverá la palabra actual.
- Operadores != y ==, que comprobarán si el iterador actual es igual o no al que se pasa como argumento (es decir, si los iteradores internos apuntan al mismo nodo).
- Operador ++, que avanzará el iterador interno hasta que se encuentre el siguiente nodo en el que termina una palabra válida. Aprovecharemos el nivel que tiene almacenado el iterador del árbol para ir formando la palabra por la que estamos iterando. Antes de avanzar el iterador del árbol, se guarda el nivel en el que nos encontrábamos. Tras mover el iterador al siguiente nodo, se comprueba cómo ha cambiado el nivel. Si el nivel ha aumentado en

una unidad, significa que hemos descendido un nivel, y por tanto, añadimos la nueva letra a la palabra que llevamos dentro del iterador. Si el nivel se ha mantenido, significa que hemos visitado un hermano, en cuyo caso sustituimos la última letra de la palabra actual por la del nuevo nodo. Si el nivel es menor, significa que estamos subiendo hacia los padres y tenemos que borrar letras de la palabra.

Finalmente, tenemos que implementar la función begin, que apunta a la primera palabra del diccionario; y la función end, que apunta a la palabra vacía al final del árbol. **Tendréis que implementar el iterador del diccionario como parte de la práctica.**

Programa de prueba - `diccionario.cpp`

Para comprobar el funcionamiento correcto del iterador implementado, se proporciona implementado el archivo diccionario.cpp. El funcionamiento de este archivo es simple:

- Se carga el contenido del archivo que se pasa como argumento en un diccionario con estructura de árbol
- Se recorre el diccionario creado con un iterador, y se imprimen por pantalla todas sus palabras

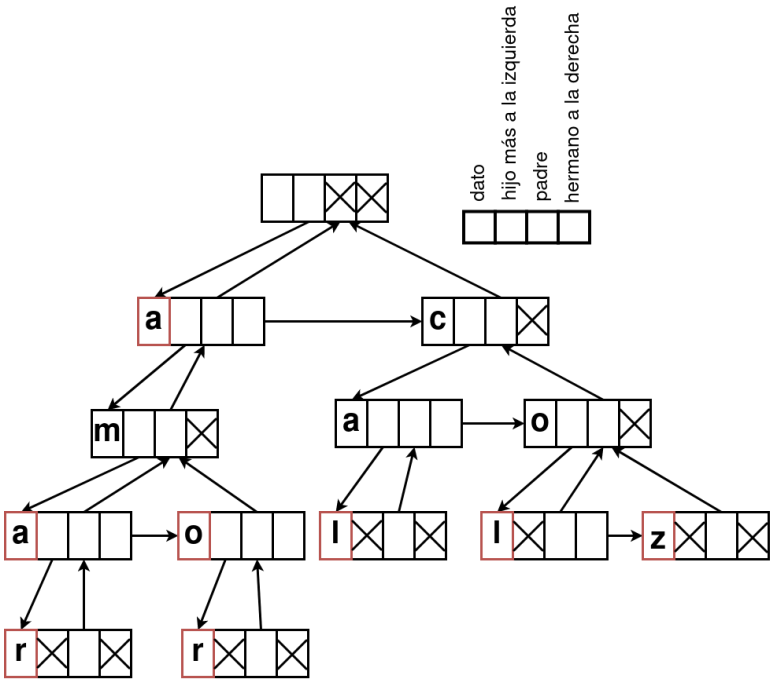
Si el iterador anterior está correctamente implementado, el programa deberá funcionar con normalidad.

En los siguientes apartados, pasamos a comentar las mejoras de eficiencia que esta implementación nos produce en el árbol.

2.3 Mejora de la eficiencia en espacio para el diccionario al usar árboles

Como hemos visto en el ejemplo del principio de la práctica, necesitamos guardar muchos menos caracteres con esta representación que con la representación anterior, pero ¿hasta qué punto hemos reducido el número de caracteres que almacenamos? Para dar respuesta a esa pregunta, vamos a implementar cierta funcionalidad sobre nuestro árbol para contar el número de veces que guardamos cada letra en nuestra estructura de árbol, y el número de veces que se utiliza cada una de estas letras para formar palabras. **Esta será una de las partes de implementación que tendréis que realizar para esta práctica.**

Como hemos dicho, vamos a diferenciar entre dos conteos distintos para cada letra en nuestro diccionario. Nos referiremos a dichos conteos como número de apariciones y número de usos, cuya diferencia se describe a continuación. Recuperamos el ejemplo del principio para ver las diferencias entre un concepto y otro:



Número de ocurrencias: Definimos el número de ocurrencias de una letra como el número de veces que aparece esa letra en nuestro diccionario, es decir, el número de nodos que tienen ese carácter almacenado. En el ejemplo anterior, el número de apariciones de la letra “a” es 3. Para contar dichas apariciones, simplemente tenemos que recorrer todos los nodos del árbol, llevando un contador con el número de apariciones de la letra, que aumentaremos en uno cada vez que visitemos un nodo que tenga ese carácter almacenado.

Número de usos: El número de usos de una letra se define como el número de veces que esa letra se utiliza para formar palabras de nuestro diccionario. Si recordáis la práctica 3, el número de veces que se usaba una letra en el diccionario coincidía con el número de veces que aparecía en el mismo, porque nunca se usaba una misma aparición de la letra para formar dos palabras distintas. Aquí, el número de apariciones será mucho menor que el número de usos. En particular, teniendo en cuenta que en nuestro diccionario están almacenadas las siguientes palabras:

a, ama, amar, amo, amor, cal, col, coz

La letra “a” se utiliza 8 veces.

Se tendrán que implementar las siguientes funciones para dotar a nuestro diccionario de la funcionalidad necesaria para hacer estos conteos:

Ejercicio 2.1 Conteo de ocurrencias en el árbol

```
public: int getOccurrences(char c)
```

Esta función debe devolver el número de ocurrencias del carácter que se pasa como argumento en nuestro árbol. Para implementar esta función de forma recursiva, necesitaremos una función auxiliar, que implementaremos como privada, y que recibirá como parámetros el carácter con el que estamos trabajando y el nodo en el que nos encontramos:

```
private: int getOccurrences(node current_node, char c)
```

Dicha función deberá contar las ocurrencias de la letra en el subárbol que cuelga del nodo `current_node`. Para ello, se deberá comprobar si la letra que hay almacenada en dicho nodo es el carácter con el que estamos trabajando, y a continuación llamar de forma recursiva a la misma función sobre el hijo a la izquierda y el hermano a la derecha de dicho nodo (siempre que no sean nulos). Finalmente, se suman los tres valores y se devuelven.

Una vez tenemos esta función auxiliar implementada, la función pública que tenemos al principio simplemente se tendrá que encargar de llamar a la función auxiliar sobre la raíz del árbol.

Ejercicio 2.2 Conteo de usos en el diccionario

```
public: int getTotalUsages(char c)
```

Esta función se encarga de contar el número de usos de la letra para formar palabras en el diccionario. De nuevo, vamos a resolver este problema utilizando una función auxiliar que trabaje de forma recursiva.

```
private: pair<int, int> getTotalUsages(node curr_node, char c)
```

Esta función nos va a devolver una pareja de valores, el número de ocurrencias del carácter en los nodos que cuelgan del nodo actual, y el número de palabras que terminan por debajo del nodo actual. La función, dado el nodo actual y el carácter que estamos buscando, realiza los siguientes pasos:

- Inicialmente, el número de ocurrencias del carácter y el número de palabras que terminan por debajo del nodo actual valen 0.
- Se llama de forma recursiva a la función sobre el hermano a la derecha y el hijo a la izquierda si estos no son nulos. Se almacenan los resultados parciales por separado.
- El número de usos es la suma del número de usos de nuestro hermano a la derecha y nuestro hijo a la izquierda, y el número de palabras completas es la suma del número de palabras completas en nuestro hermano a la derecha y nuestro hijo a la izquierda.
- Si en el nodo actual el carácter que tenemos coincide con el carácter que estamos buscando, aumentamos el número de usos de la letra en tantas unidades como palabras terminen en nuestro hijo de la izquierda.
- Si en el nodo actual termina una palabra, tendremos que aumentar el número de palabras completas. Si además el carácter del nodo actual coincide con el buscado, el número de usos de la letra se aumenta en 1.
- Finalmente, devolvemos el resultado.

La función pública que utilizamos, simplemente llama a la función anterior sobre la raíz del árbol, y devuelve el primer valor del par (ya que el segundo nos sirve para hacer cálculos intermedios en la función privada, pero no nos da información de cara a los usuarios).

IMPORTANTE: La solución ha de ser recursiva. No obstante, no es necesario que implementéis estas funciones de la forma que se describe anteriormente. El pseudocódigo anterior corresponde a las soluciones que nosotros hemos implementado. Si alguien piensa en otra forma válida de hacer estos conteos, estaremos encantados de discutir dicha solución. No obstante, este es un ejercicio para practicar la implementación de funciones recursivas en árboles. **Cualquier implementación no recursiva, aunque devuelva un resultado correcto, no será considerada como válida.**

Programa de prueba - `cantidad_letras.cpp`

Para comprobar el funcionamiento correcto de las funciones de conteo de letras implementadas (tanto para el número de ocurrencias como para el uso), debéis implementar un pequeño programa que las utilice. Dicho programa se llamará `cantidad_letras.cpp`, y su funcionamiento será el siguiente: Recibirá como argumentos un fichero de diccionario y un fichero de letras (podéis usar la implementación de LettersSet de la práctica anterior, debería ser compatible con esta práctica), creará el LettersSet y el Dictionary con la información de dichos ficheros, y para cada letra del diccionario calculará el número de veces que se usa esa letra y el número de ocurrencias de la letra en la estructura de diccionario. En el diccionario de ejemplo con el que venimos trabajando en este guión, la salida debería ser la siguiente:

Letra	Usos	Ocurrencias
A	8	3
C	3	1
L	2	2
M	4	1
R	2	2
O	4	2
Z	1	1

3.- Resolución del juego de las letras: Solver

En la práctica anterior se desarrollaron todos los TDAs necesarios para resolver el juego de las letras. En la primera parte de esta práctica, se ha explorado el uso de un contenedor alternativo (un árbol LCRS) para implementar el TDA Dictionary, comprobándose la mejora que supone en memoria. Ahora que tenemos todas estas estructuras de datos, estamos listos para resolver el juego de las letras.

Un *solver* es un sistema que resuelve un problema de forma automática. En nuestro caso, vamos a programar un *solver* para el juego de las letras. Este *solver* tiene que ser capaz de dar la/s mejor/es solución/es dado un diccionario, un conjunto de letras, un modo de juego, y un número de letras.

3.1 - Primera aproximación a la solución del problema

En primer lugar, daremos una solución al problema de forma sencilla utilizando el iterador sobre el diccionario que implementamos anteriormente. Recorreremos el diccionario y comprobaremos qué palabras de las almacenadas podemos formar con las letras disponibles. Para dicho conjunto de palabras, calcularemos sus puntuaciones y nos quedaremos con la mejor solución o mejores soluciones.

Ejercicio 3 - Solver inicial

Esta clase Solver va a hacer uso de los TDAs creados en esta práctica y la anterior, así como del iterador de palabras del diccionario. En concreto, vamos a usar el TDA Dictionary para saber el conjunto de palabras que consideramos soluciones posibles de nuestro juego, el TDA LettersSet para saber cuántas letras de cada tipo disponemos y la puntuación que nos da cada una al usarla en la solución, y el TDA LettersBag (que por debajo utiliza el TDA Bag) las letras concretas con las que jugaremos una partida. Nuestro Solver tiene que ofrecer la mejor solución para una partida y modo de juego determinado. Como esta solución no tiene por qué ser única (puede haber varias palabras con una misma puntuación), necesitamos poder ofrecer un conjunto de palabras con una misma puntuación.

En concreto, nuestra clase ofrecerá las soluciones en un vector de la STL de la forma a través del método `getSolutions()` (**IMPORTANTE: Este pair no es un dato miembro del TDA Solver**):

```
pair<vector<string>, int>
```

Como estructura de datos para guardar la palabra o palabras que constituyan una solución se propone usar un vector de strings. El vector de strings contendrá las mejores soluciones posibles que se puedan formar con las letras de las que se disponga en la partida, y el entero será la puntuación de todas esas palabras (es suficiente con devolver la puntuación una sola vez porque todas las palabras deberán tener la misma puntuación, ya que son las mejores soluciones).

Dentro de este TDA, queremos implementar la siguiente funcionalidad:

- Constructor por parámetros. La clase Solver debe conocer el diccionario de palabras permitidas y el LettersSet con la información de las letras para poder formar sus soluciones. Como esta información es inmutable para todas las partidas que se jueguen, tiene sentido incluirlas como atributos dentro de la clase cuando se cree un objeto de tipo Solver.

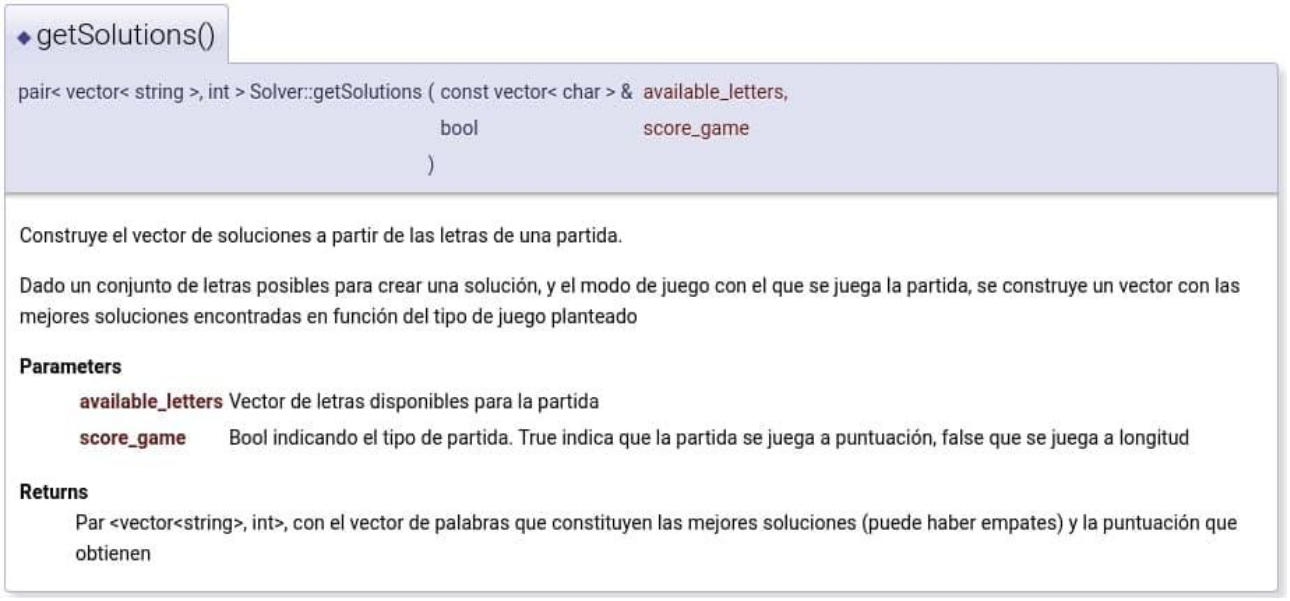
◆ Solver()

```
Solver::Solver ( const Dictionary &  dict,
                 const LettersSet &  letters_set
               )
```

Constructor con parámetros.

Crea un Solver que tiene asociado un Dictionary y un LettersSet

- Función para obtener las mejores soluciones dado el vector de letras disponibles y el tipo de partida que se juega. Como podríamos jugar con un mismo Solver varias partidas consecutivas, y en cada partida jugaríamos un tipo de juego (puntuación o longitud) y con unas letras disponibles distintas, tiene sentido que esta información se pase como parámetro a la función que calcula dichas soluciones.



◆ getSolutions()

```
pair< vector< string >, int > Solver::getSolutions ( const vector< char > &  available_letters,
                                                     bool                    score_game
                                                   )
```

Construye el vector de soluciones a partir de las letras de una partida.

Dado un conjunto de letras posibles para crear una solución, y el modo de juego con el que se juega la partida, se construye un vector con las mejores soluciones encontradas en función del tipo de juego planteado

Parameters

available_letters Vector de letras disponibles para la partida

score_game Bool indicando el tipo de partida. True indica que la partida se juega a puntuación, false que se juega a longitud

Returns

Par <vector<string>, int>, con el vector de palabras que constituyen las mejores soluciones (puede haber empates) y la puntuación que obtienen

Uso de funciones auxiliares: La función anterior es un claro ejemplo de función en el que el uso de funciones auxiliares puede mejorar la legibilidad del código. Por tanto, se valorará positivamente el uso de dichas funciones a la hora de implementar esta solución.

Programa de prueba - `partida_letras.cpp`

Para probar el funcionamiento de nuestro TDA, vamos a implementar un programa que recibirá cuatro argumentos:

1. Ruta al archivo que contiene el diccionario
2. Ruta al archivo que contiene las letras y su puntuación
3. Modo de juego (L = longitud, P = puntuación)
4. Cantidad de letras para la partida

Dicho programa se encargará de construir el LettersSet para la partida a partir del archivo de letras, el Dictionary con las palabras que se consideran soluciones correctas, el Solver que va a jugar la partida y la LettersBag que se utilizará para extraer las letras. Una vez construidos los TDAs necesarios, extraerá las letras con las que se jugará la partida, utilizará el Solver para buscar las soluciones, e imprimirá por pantalla tanto las letras para la partida como las mejores soluciones que se pueden obtener con dichas letras y la puntuación de dichas soluciones.

A modo de ejemplo, la salida del programa debe ser la siguiente:

```
> ./build/partida_letras data/diccionario.txt data/letras.txt L 9

LETRAS DISPONIBLES:
D S N T D A I E N
SOLUCIONES:
dentina
entidad
sentina
PUNTUACION:
7
```

IMPORTANTE: El formato de la salida es especialmente relevante. El juez se basa en la salida correctamente formateada de vuestro programa para evaluar la solución. Para comprobar que la salida de vuestra solución es correcta, se utilizan las letras que ha generado vuestro programa para buscar las mejores palabras con nuestro Solver. Después, se comprueba que ambos han obtenido las mismas soluciones y con la misma puntuación.
Si vuestra salida no está correctamente formateada el Juez devolverá `Unexpected Error`.

3.2 - Mejora de la eficiencia en el juego de las letras

La solución del apartado anterior consistía en iterar sobre el Diccionario y comprobar qué palabras se podían formar con las letras que teníamos disponibles. De esta forma, al final, el Solver se quedaba sólamente con aquellas palabras que se podían formar, y de ahí seleccionamos las de mayor puntuación como solución final.

El problema de esta solución es que nos obliga a iterar por el árbol completo, y descartar a posteriori las palabras que no podemos formar con las letras que tenemos disponibles. Es mucho más eficiente recorrer solamente aquellas ramas del árbol que nos van a permitir formar palabras, ignorando aquellas ramificaciones que contengan letras que no tenemos disponibles. De esta manera, en lugar de iterar por todo el árbol descartando las palabras que no podemos formar con un método externo, lo que vamos a hacer es pedirle al diccionario directamente la lista de palabras que podemos formar. Para ello, nos vamos a apoyar en la definición de un nuevo iterador `possible_words_iterator`. Este iterador se va a encargar de recorrer la estructura del diccionario explorando sólo las ramas de la estructura que pueden generar palabras válidas dada una bolsa de letras.

Ejercicio 4 - Iterador de palabras válidas

Los métodos que se piden implementar para este iterador son los siguientes:

- Funciones begin y end que apuntan al principio y fin de la estructura.
- Constructores por defecto y por parámetros, recibiendo este último el nodo actual y bolsa de letras disponible.
- Constructor de copia.
- Operadores de asignación, operadores == y != que comprobarán si dos iteradores son iguales o distintos entre sí.
- Operador * que nos devolverá la palabra válida actual.
- Operador ++ que avanzará el iterador a la siguiente palabra válida.

Para la construcción de este iterador no se puede emplear el iterador del Ejercicio 1. El objetivo de este nuevo iterador es recorrer el árbol de forma más eficiente, descartando la exploración de ramas que no producen palabras válidas según nuestra bolsa de letras disponibles. La idea del iterador de palabras válidas es realizar un recorrido del árbol en preorden pero podando aquellas ramas que en las que no se podrá formar una palabra válida por falta de letras.

Para ello tendremos que mantener en memoria la bolsa de letras disponibles en el momento actual. Es importante recalcar que esta bolsa irá cambiando conforme nuestro iterador descarte o genere palabras disponibles. Al descender en el nivel del árbol tendremos menos letras disponibles y si ascendemos en el nivel del árbol debemos retornar las letras usadas al conteo de letras disponibles.

Internamente el iterador mantendrá un multiset con las letras disponibles, una string que mantiene la palabra actual y el nodo que actualmente estamos visitando. La cadena de texto nos servirá a modo de pila para ir construyendo la siguiente palabra del iterador. De esta forma podremos apilar o eliminar la letra del último nodo visitado.

Programa de prueba - `palabras_validas_diccionario.cpp`

Para comprobar el funcionamiento correcto del iterador de palabras válidas, se proporciona el archivo palabras_validas_diccionario.cpp. El funcionamiento de este archivo es simple:

- Se carga el contenido del archivo que se pasa como argumento en un diccionario con estructura de árbol.
- El resto de argumentos que se pasan en la llamada al programa son las letras disponibles que se utilizan para formar palabras.

- Se recorre el diccionario creado con un iterador, y se imprimen por pantalla sus palabras válidas según las letras especificadas.

Si el iterador anterior está correctamente implementado, el programa deberá funcionar con normalidad, devolviendo las palabras válidas en orden alfabético.

Ejercicio 5 - Solver eficiente

Tras haber implementado el nuevo iterador podemos programar un nuevo Solver que ataque el problema de las letras con mejor eficiencia en tiempo que el anterior. En el anterior debíamos iterar por todas las palabras disponibles del diccionario y comprobar a posteriori su validez, en este caso evitamos generar palabras no válidas y por tanto habremos reducido el número de elementos explorados.

En esta sección se debe replicar, con la misma interfaz, el Solver anterior teniendo de nuevo el mismo constructor y la función `getSolutions`.

Programa de prueba - `partida_letras_eficiente.cpp`

Para probar el funcionamiento de nuestro TDA, vamos a implementar un programa que recibirá cuatro argumentos:

5. Ruta al archivo que contiene las letras y su puntuación
6. Ruta al archivo que contiene el diccionario
7. Modo de juego (L = longitud, P = puntuación)
8. Cantidad de letras para la partida

Dicho programa se encargará de construir el LettersSet para la partida a partir del archivo de letras, el Dictionary con las palabras que se consideran soluciones correctas, el Solver que va a jugar la partida y la LettersBag que se utilizará para extraer las letras. Una vez construidos los TDAs necesarios, extraerá las letras con las que se jugará la partida, utilizará el Solver para buscar las soluciones, e imprimirá por pantalla tanto las letras para la partida como las mejores soluciones que se pueden obtener con dichas letras y la puntuación de dichas soluciones.

A modo de ejemplo, la salida del programa debe ser la siguiente:

```
> ./build/partida_letras_eficiente data/diccionario.txt data/letras.txt L
9

LETRAS DISPONIBLES:
D S N T D A I E N
SOLUCIONES:
dentina
entidad
sentina
PUNTUACION:
7
```

IMPORTANTE: El formato de la salida es especialmente relevante. El juez se basa en la salida correctamente formateada de vuestro programa para evaluar la solución. Para comprobar que la

salida de vuestra solución es correcta, se utilizan las letras que ha generado vuestro programa para buscar las mejores palabras con nuestro Solver. Después, se comprueba que ambos han obtenido las mismas soluciones y con la misma puntuación.
Si vuestra salida no está correctamente formateada el Juez devolverá `Unexpected Error`.

4.- Comparativa de eficiencia entre los Solvers

Tras haber proporcionado dos Solvers al problema de las letras queda la duda de hasta qué punto es más eficiente el segundo de ellos. Para ello planteamos enfrentar ambas soluciones (correctas ellas dos) con la intención de comprobar empíricamente el grado de mejora.

Desde un punto de vista teórico es fácil deducir que el número de palabras que nos devuelve el iterador de palabras válidas es menor o igual que el número de palabras devuelto por el iterador del diccionario. Por tanto, en el peor de los casos ambos tienen la misma eficiencia, no siendo así en el resto de los casos al tener que comprobar menos palabras.

Para analizar este fenómeno se propone el estudio del tiempo consumido por un Solver y otro para proporcionar la mejor o las mejores palabras en cuanto a puntuación. Se recomienda para este uso la librería `<chrono>` y emplear la función `high_resolution_clock::now()` para medir los segundos consumidos hasta dar la solución final. Se puede consultar la documentación de dicha clase en: https://cplusplus.com/reference/chrono/high_resolution_clock/.

Ejercicio 6 - Análisis de eficiencia de las soluciones:

Para este ejercicio se pide un análisis empírico **lo más completo posible** de la eficiencia del algoritmo. A modo de indicación, las variables que influyen en el tiempo de ejecución de este algoritmo son el número de letras con las que se juega una partida, y el tamaño del diccionario.

4.- Práctica a Realizar

Atendiendo a la explicación de la práctica de los apartados anteriores, hay seis ejercicios a resolver:

1.- Implementación del iterador diccionario (versión árbol):

Implementar un iterador de palabras de la nueva versión de la clase dictionary y comprobar su correcto funcionamiento con el programa `diccionario.cpp`.

2.- Implementación de los contadores de usos y ocurrencias de las letras:

Para este ejercicio, se deben implementar los contadores de uso y ocurrencias para las letras del diccionario, junto con el programa `cantidad_letras.cpp` que compruebe su correcto funcionamiento.

3.- Implementación del solver inicial:

En este ejercicio se pide implementar un Solver para el problema de las letras empleando el iterador de palabras del diccionario anteriormente realizado, junto con el programa `partida_letras.cpp` que compruebe su correcto funcionamiento.

4.- Implementación del iterador de palabras válidas (versión árbol):

Implementar un iterador de palabras válidas dada una bolsa de letras para la clase dictionary, junto con el programa `palabras_validas_diccionario.cpp` que compruebe su correcto funcionamiento.

5.- Implementación del solver eficiente:

En este ejercicio se pide implementar un Solver para el problema de las letras empleando el iterador de palabras válidas dada una bolsa de letras anteriormente realizado, junto con el programa `partida_letras_eficiente.cpp` que compruebe su correcto funcionamiento.

6.- Análisis de eficiencia de las soluciones:

Para este ejercicio se pide un análisis empírico **lo más completo posible** de la eficiencia del algoritmo. A modo de indicación, las variables que influyen en el tiempo de ejecución de este algoritmo son el número de letras con las que se juega una partida, y el tamaño del diccionario.

5.- Documentación y entrega

Toda la documentación de la práctica se incluirá en el propio Doxygen generado, para ello se utilizarán tanto las directivas Doxygen de los archivos .h y .cpp como los archivos .dox incluídos en la carpeta doc.

1. Código:
- Se dispone de la siguiente estructura de ficheros:
  - /
    - estudiante/ (Aquí irá todo lo que desarrolle el alumno)
      - doc/ (Imágenes y documentos extra para Doxygen)
      - include/ (Archivos cabecera)
      - src/ (Archivos fuente)
    - data/ (Datos de ejemplo)
    - data_eficiencia/ (Datos de ayuda para el ejercicio final de eficiencia)
    - CMakeList.txt (Instrucciones CMake)
    - Doxyfile.in (Archivo de configuración de Doxygen)
    - juez.sh (Script del juez online [**HAY QUE CONFIGURARLO**])
- Es importantísimo leer detenidamente y entender qué hace CMakeList.txt.
- El archivo Doxyfile.in no tendríais por qué tocarlo para un uso básico, pero está a vuestra disposición por si queréis añadir alguna variable (no cambiéis las existentes)
- juez.sh crea un archivo submission.zip que incluye ya TODO lo necesario para subir a Prado a la hora de hacer la entrega. No tenéis que añadir nada más, especialmente binarios o documentación ya compilada.

4. Elaboración y puntuación
- La práctica se realizará POR PAREJAS. Los nombres completos se incluirán en la descripción de la entrega en Prado. Cualquiera de los integrantes de la pareja puede subir el archivo a Prado, pero SOLO UNO.
- Desglose:
  - **(0.2)** Ejercicio 1 - Implementación del iterador de la clase Dictionary.
  - **(0.2)** Ejercicio 2 - Implementación de los contadores de letras.
  - **(0.1)** Ejercicio 3 - Implementación del Solver inicial.
  - **(0.3)** Ejercicio 4 - Implementación iterador de palabras válidas.

    - **(0.1)** Ejercicio 5 - Implementación del Solver eficiente.
    - **(0.1)** Ejercicio 6 - Comparativa de eficiencia entre los Solvers.
- La documentación de esta práctica, aunque no tiene nota concreta asociada, se deberá entregar de la misma forma que en las prácticas anteriores. El no entregar el código documentado puede suponer una penalización de la nota final de la práctica de hasta **0.25 puntos**.
- La fecha límite de entrega aparecerá en la subida a Prado.