UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Práctica Final: Cifras y Letras

Práctica 5: 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 Cifras y letras

En esta práctica, y las prácticas sucesivas, nos centraremos en el juego conocido como cifras y letras. Este juego, que se ha popularizado a través de concursos de televisión en distintos países. Nos centraremos de momento en la prueba de las letras. Exploraremos el juego de las cifras en las siguientes prácticas, debido a su mayor complejidad

Prueba de las letras

Esta parte del juego consiste en 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 y las prácticas siguientes construiremos las estructuras de datos adecuadas para resolver el problema de las letras, así como programas que nos permitan obtener la solución a una partida.

Estaremos especialmente interesados en dar una buena solución al problema de las letras. En esta primera práctica, estudiaremos qué información necesitamos almacenar, y cuál es la forma más adecuada de almacenarla, para poder jugar una partida al juego de las letras. En las prácticas siguientes, estudiaremos dos formas distintas de resolver el juego, que dependerá en parte de la estructura de datos subyacente.

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

Teniendo en cuenta la descripción del juego de las letras que hemos hecho en el apartado anterior, parece claro que vamos a necesitar tres almacenes principales de información para poder jugar una partida de las letras

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 poder aprovecharla para calcular las puntuaciones de nuestras palabras. Esta estructura de datos se denominará Conjunto de Letras (TDA LettersSet)

Además, como hemos dicho que necesitaremos extraer letras aleatorias para una partida, utilizando para ello las repeticiones del archivo anterior, construiremos también un contenedor específico para ello. 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)

Información sobre las palabras

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
aarónico
aba
ababa
ababillarse
ababol
abacal
abacalero
...
```

Fichero 2: diccionario.txt

Y necesitaremos una estructura de datos que nos permita consultar este listado de palabras desde nuestro programa. Este contenedor será nuestro TDA Dictionary.

Debido al tipo de información con la que tenemos que trabajar, los contenedores lineales que hemos utilizado hasta ahora no parecen la mejor solución. Ahora, tendremos que trabajar con información que tiene una estructura muy específica, y el orden de inserción de los elementos no parece la forma más cómoda de trabajar. Por este motivo, vamos a utilizar lo que se conoce como contenedores no lineales, o contenedores asociativos.

2.- Resolución del juego de las letras: Solver

En esta práctica trabajaremos con lo que se conoce como "solver". Un *solver* es un sistema que resuelve un problema de forma automática. En nuestro caso, vamos a programar un solver para el problema 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.

2.1 Solver

Esta clase va a utilizar las estructuras de datos que hemos creado en la práctica anterior para ofrecer soluciones al juego de las letras. Esta estructura va a hacer uso de todos los TDAs creados en la práctica 4. 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()` (**OJO! Este pair no es un dato miembro del TDA Solver**):

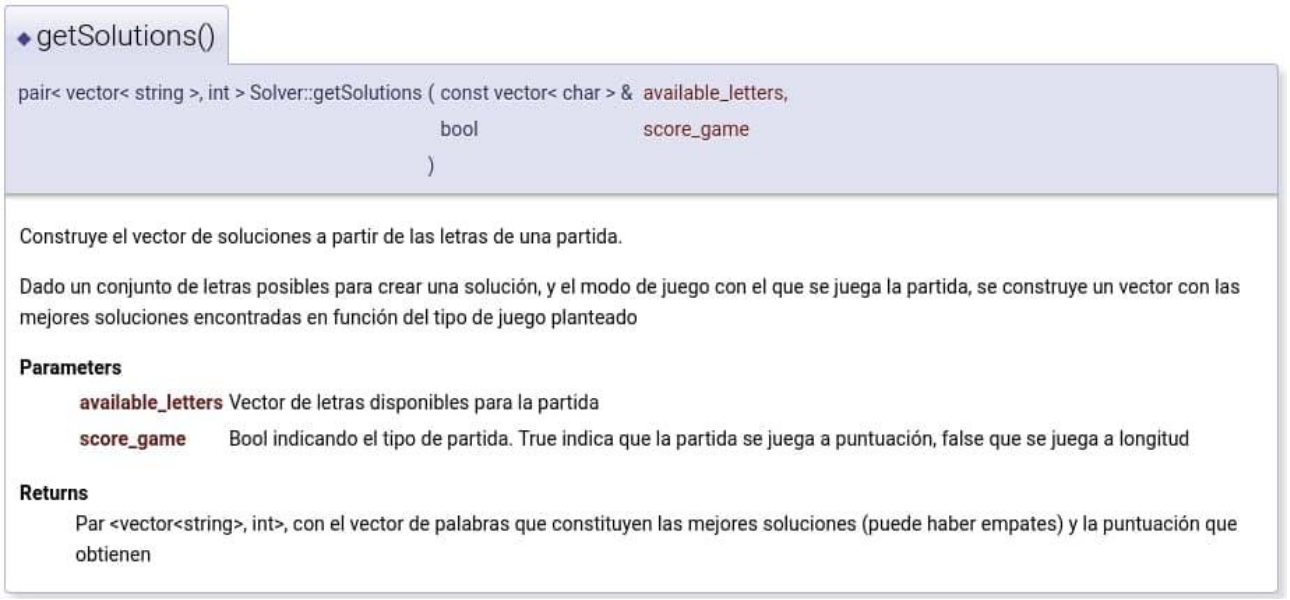
```
pair<vector<string>, int>
```

El set 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. El 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.

- 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

Importante - Uso de funciones auxiliares: La función anterior es un claro ejemplo de función en el que la construcción 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 las letras y su puntuación
2. Ruta al archivo que contiene el diccionario
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/letras.txt data/diccionario.txt L 9

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

(OJO! El formato de la salida es especialmente importante). 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.- Práctica a Realizar

1.- Ejercicio obligatorio: Construcción del Solver

Se propone como único ejercicio la implementación del **Solver**, que implementa el *solver* que hemos descrito en los apartados anteriores. Para ello, será necesario:

1. Construir la clase *Solver* a partir de los métodos especificados en este documento.
2. Construir el programa ejecutable asociado para comprobar el comportamiento del solver.

4.- 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. La documentación debe incluir:
- TODOS los métodos y clases debidamente documentados con la especificación completa. Se valorará positivamente descripciones exhaustivas.
    - Todos los métodos del Solver así como los métodos utilizados de otros TDAs **DEBEN DOCUMENTAR LA COMPLEJIDAD COMPUTACIONAL O-grande** (Big-O) en el Doxygen generado.
- Las dos tareas de la práctica tienen un ejecutable asociado que debe ser descrito en la página principal o en una página *ad hoc*.
- Todos las páginas de la documentación generada deben ser completas y estar debidamente descritas.

2. A considerar:
- Deben respetarse todos los conocimientos adquiridos en los temas de Abstracción y Eficiencia. En especial el **principio de ocultamiento de información** y las distintas estrategias de **abstracción**.
- Una solución algorítmicamente correcta pero que contradiga el punto anterior será considerada como errónea.
- Una solución algorítmicamente correcta pero que no utilice el contenedor subyacente requerido en cada caso (es decir, debemos implementar Solver utilizando un set), también será considerada como errónea.

3. 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)

        - 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:
        - **(0.60)** Ejercicio 1
            - **(0.30)** Implementación
            - **(0.30)** Documentación
    - La fecha límite de entrega aparecerá en la subida a Prado.