Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Departamento de Ciencias de la Computación

Analizador Sintáctico

R212 - Estructuras de Datos y Algoritmos I

Trabajo práctico final

Alumno:
Sferco, Martín

Septiembre, 2023

Índice

1. Introducción

En este Trabajo Práctico final de la materia, se nos propuso realizar un *analizador sintáctico* o *parser*. Dado un archivo con líneas de caracteres del abecedario (pudiendo ser minúsculas, mayúsculas, o una mezcla entre ellas), se encarga de colocar espacios de manera que la frase resultante esté formada por palabras aceptadas. Dichas palabras aceptadas se encuentran en un diccionario de palabras válidas. Además, el analizador sintáctico se encargará de devolvernos una separación de palabras que maximice en cada paso, el tamaño de las mismas.

En el caso de que nos encontremos en un punto donde no es posible formar ninguna palabra válida, se saltea uno o más caracteres (que ahora consideraremos como errores) para continuar con el análisis del resto de la frase.

A continuación, hablaremos sobre las estructuras de datos utilizadas y cómo construimos el algoritmo en base a relacionar dichas estructuras y sus funcionalidades.

2. Estructuras de Datos

Para la realización del trabajo, nos apoyamos principalmente en dos estructuras de datos, y una adicional que deriva de una de las otras dos. A continuación, explicamos un poco por qué decidimos elegirlas y describimos su funcionamiento básico.

2.1. CTrie - Compact Trie

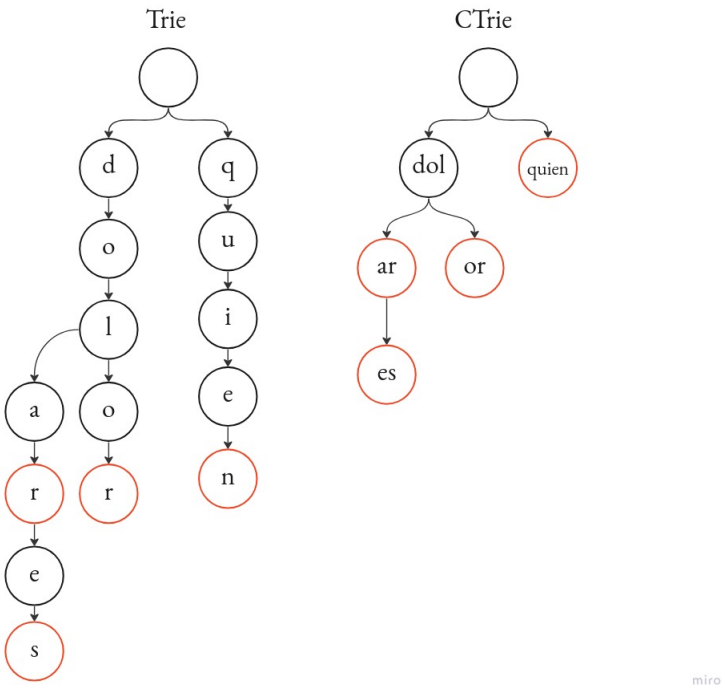
2.1.1. Elección del CTrie

Al comenzar a pensar en cómo sería el algoritmo, tratamos de ir leyendo la cadena e ir quedándonos con las palabras que habían coincidido con todas las letras que había leído hasta ese momento. Es decir, estábamos quedándonos con las palabras que tenían como prefijo la parte de la cadena que había leído. Fue entonces que se comenzó a averiguar sobre lo que se conocen como *Árboles de Prefijos*.

Terminamos utilizando lo que se conoce como un `Compact Trie`, la cual es una forma optimizada de un `Trie`. Un `Trie` es una estructura de tipo árbol de búsqueda k-ario (tiene k hijos), generalmente utilizada para almacenar cadenas. Lo interesante es que la posición de un nodo en el árbol está asociada a un prefijo de una de las cadenas ingresadas, y todos los hijos de un nodo tienen como prefijo común la clave asociada a dicho nodo. Esto último nos fue muy útil, ya que es lo que estábamos haciendo en nuestra primera versión del algoritmo.

El `Trie` no nos resultaba tan eficiente, ya que cuando las palabras del diccionario no tenían muchos prefijos en común, el árbol se bifurcaba demasiado y había una gran utilización de nodos para representar cada prefijo, sumado a que cada nodo tenía un total de 26 hijos (uno por cada letra del abecedario en minúscula). Esto se traducía en un gran gasto de memoria.

El `CTrie` optimiza la cantidad de espacio utilizado de la siguiente manera: todos los nodos consecutivos que tengan uno o ningún hijo y no sean fin de palabra, se compactan para formar un único nodo. Por ejemplo, si en nuestro diccionario tuviésemos las palabras `dolar, dolares, dolor, quien`, se generarían los siguientes árboles:



siendo cada nodo de color rojo el fin de una palabra válida del diccionario.

Es importante aclarar que en el peor caso, que sería un diccionario con palabras y todos los prefijos propios de las mismas, el `CTrie` y el `Trie` tienen la misma complejidad espacial.

2.1.2. Estructura del CTrie

El tipo `CTrie` es básicamente un puntero a una estructura `CTrieNode`, la cual podemos ver a continuación:

```
1  struct CTrieNode { // Estructura de un nodo del CTrie
2
3    char* string;  // Inicio de la cadena almacenada
4
5    int length; // Largo de la cadena almacenada
6
7    unsigned int endOfWord; // Vemos si llegamos hasta un fin de cadena
8
9    unsigned int startMemoryBlock; // Vemos si en el nodo, 'string' apunta al
10                                  // comienzo de un bloque de memoria
11
12   struct CTrieNode** childs; // Hijos del nodo
13 };
```

Profundicemos sobre algunos campos de la estructura:

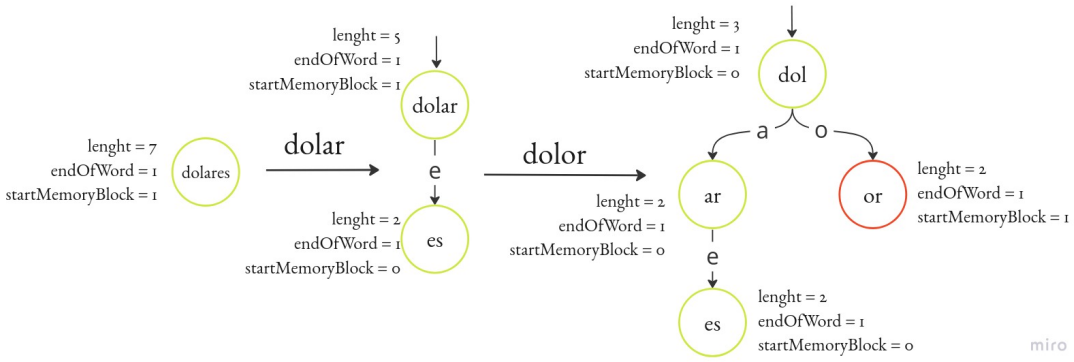
- `endOfWord`: Sirve para indicar si al leer todo el string del nodo, se llegó a un fin de palabra. Similar a lo que vimos con la imagen en donde comparamos el `CTrie` con el `Trie`.

- `startMemoryBlock`: Como veremos a continuación, en la sección de inserción de palabra en `CTrie`, una mejora que hicimos a la estructura `CTrie`, es que reutilizamos los caracteres que ya fueron cargados en memoria para utilizarlos en los campos `string` de otros nodos. Es necesario este campo, ya que cuando destruyamos la estructura, debemos liberar todos los bloques una única vez.

- `childs`: Son los hijos del nodo. Diremos a lo largo del informe el *"hijo asociado a un caracter"*, haciendo referencia a la asociación que haremos entre los índices $\{0, \ldots, 25\}$ del arreglo de hijos del nodo, con los caracteres $\{a, \ldots, z\}$. De esta manera, es más fácil ver el camino que estamos siguiendo en el árbol.

2.1.3. Inserción de palabra en CTrie

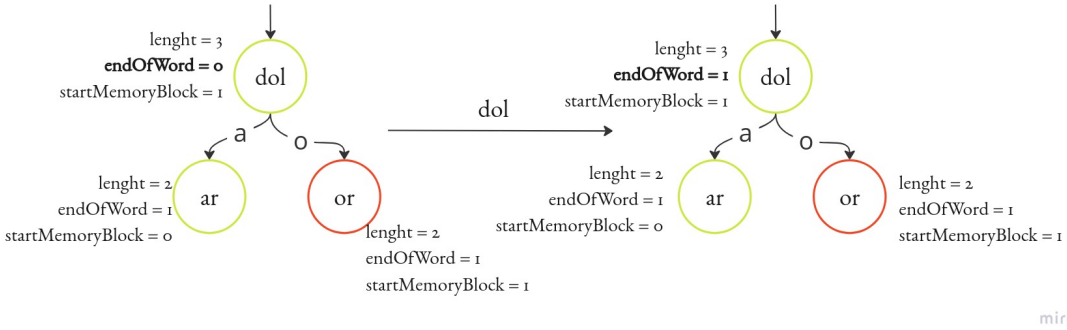
Antes de comenzar con la explicación de inserción de una palabra, explicaremos la mejora que le hicimos al `CTrie`, mencionada en los ítems anteriores. Cuando al insertar una palabra, teníamos que agregar un nodo cuyo string corresponde a un pedazo de string de algún nodo ya cargado, no copiamos físicamente los caracteres, sino que copiamos el puntero de donde debería comenzar el nuevo pedazo de string. A lo largo del informe, indicaremos ésto poniendo los nodos del mismo color, mostrando que provienen del mismo bloque de memoria. A continuación un ejemplo:



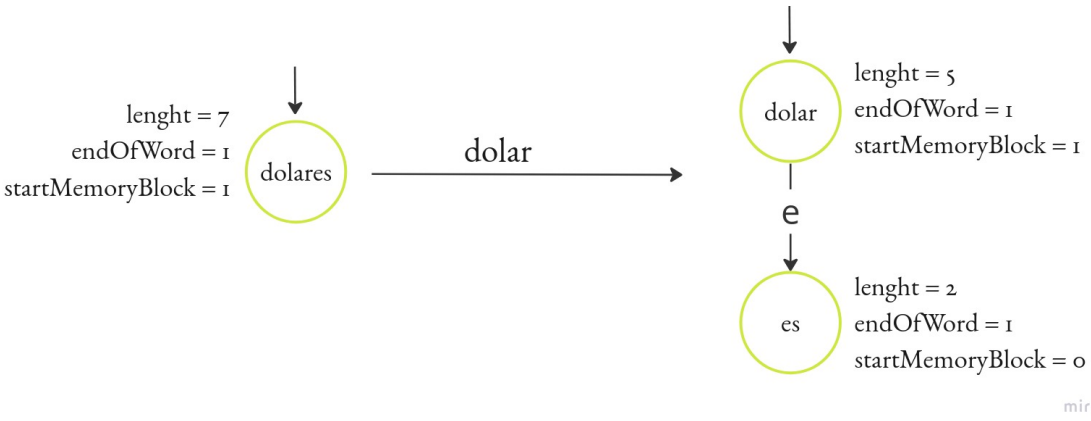
Esta optimización no sirve de mucho cuando se ingresan las palabras más chicas, y luego palabras que tienen como prefijos a las palabras ya ingresadas, ya que no estamos reutilizando ningún caracter cargado. De igual manera, ésto no empeora el rendimiento del `CTrie`; sólo es un caso en el cual no se reutiliza memoria ya cargada.

A la hora de insertar una palabra en un CTrie, nos enfrentamos con 4 casos. Indicaremos cuáles son los casos, y cómo los resolvimos:

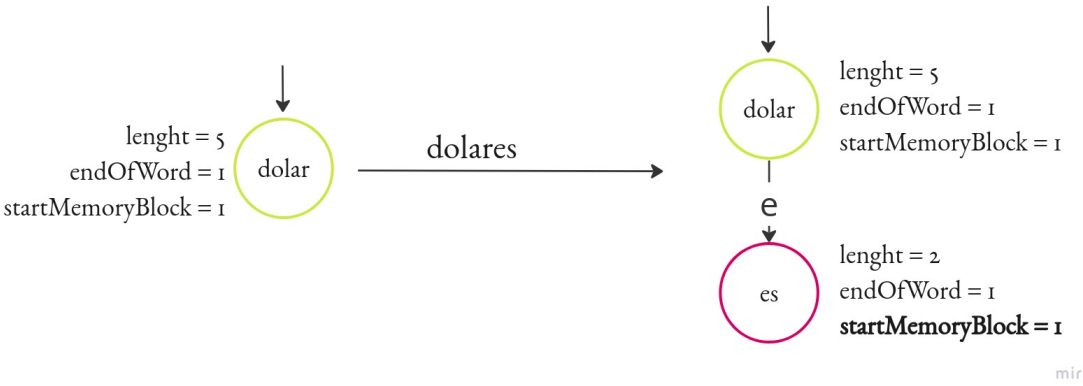
1. La palabra tenia el mismo largo que el string del nodo, y coincidieron todos sus caracteres. Se resolvió poniendo `endOfWord = 1`, ya que ese nodo, se convirtió en el fin de una palabra válida:



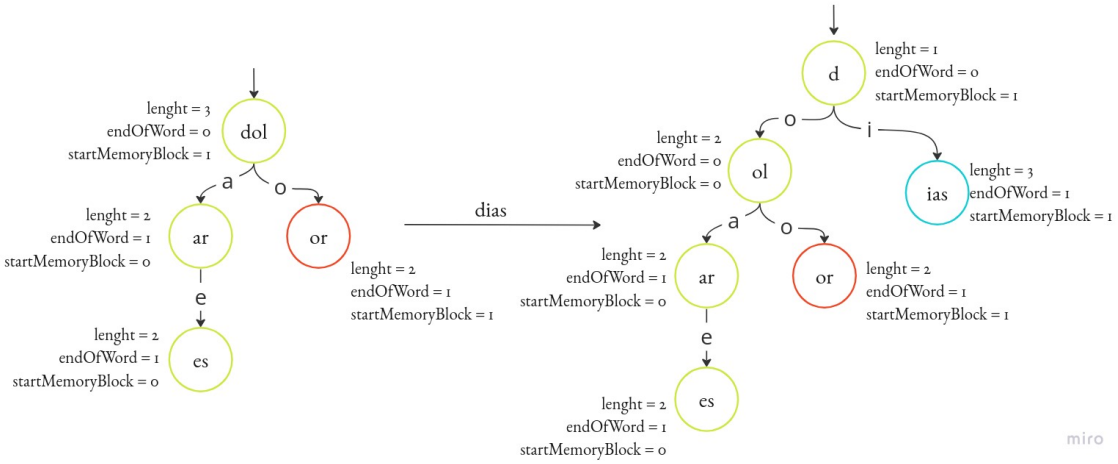
2. La palabra era más corta que el string del nodo, y coincidieron todos sus caracteres. En este caso, dividimos el nodo en dos partes. La primera parte se mantuvo en el mismo nodo, mientras que la segunda fue a un nuevo nodo cuyo string será del mismo bloque que el del nodo original. Esto lo realiza la función `ctrie_split_node()`:



3. La palabra era más larga que el string del nodo, y coincidieron todos sus caracteres. Esto se resolvió, insertando recursivamente el resto de la palabra, en el hijo asociado al primer caracter de la parte cadena que insertamos en el subarbol:



4. La palabra no coincidió con un caracter del nodo. En este caso, tuvimos que hacer una bifurcación, en donde la primera parte que sí coincidió se mantiene en el nodo, y las otras dos partes, formaron nuevos nodos, cada uno con la parte del string que no coincidieron. De esto se encarga la función `ctrie_create_bifurcation()`:



2.1.4. Destrucción del CTrie

Para la destrucción del `CTrie`, lo realizamos de manera recursiva, análogamente a como lo hacemos en los árboles binarios de búsqueda: liberamos cada uno de los hijos de manera recursiva, y luego liberamos el nodo en el que nos encontramos. La única diferencia es que solamente liberamos el string del nodo cuando `startMemoryBlock = 1`, ya que nos está indicando que nos encontramos en el comienzo del bloque.

2.2. DString - Dynamic String

2.2.1. Elección del DString

A la hora de parsear las líneas, necesitábamos poder volver atrás en la lectura del archivo en caso de que no encontremos prefijo válido, para marcar dicha posición como error, y seguir con el análisis. Algunos de los enfoques que planteamos, pero no fueron utilizados, eran los siguientes:

- **Moverse en el propio archivo**: Podíamos leer y movernos dentro del archivo utilizando las funciones `ftell()` y `fseek()`, pero eran muy costosas y hacían muy lento el programa. También averiguamos un poco sobre la estructura interna de `FILE`, pero no quisimos tratar con la propia implementación de la estructura (ya que no estaríamos respetando la abstracción de la misma) sin tener alguna función de la interfaz que nos lo permita.
- **Copiar toda la línea a un arreglo**: No resultaba eficiente tener que copiar todo a un arreglo, para luego tener que parsearlo completamente.

Finalmente, decidimos implementar un arreglo dinámico específico para `char` (en lugar de implementar un arreglo dinámico general), ya que un único caracter ocupa menos que una dirección de memoria y, al trabajar con tantos datos, resultaría mucho más costoso en memoria, almacenar las direcciones de memoria. Además, esta estructura nos permite

leer de ella cuando ya se había leído esa parte del archivo y, cuando no la habíamos leído, cargamos los nuevos caracteres desde el archivo hacia la estructura. También, una vez creado el `DString`, ya teníamos cargado en la memoria ese arreglo, por si en un futuro necesitamos guardar otra línea del archivo a parsear.

2.2.2. Estructura del DString

El tipo `DString` es básicamente un puntero a una estructura `struct _DString`, la cual podemos ver a continuación:

```
1  struct _DString { // Estructura del DString
2
3   char* chars; // Datos guardados
4   int capacity; // Capacidad total de elementos
5   int used; // Cuanto del arreglo esta siendo utilizado
6
7  };
```

Sobre los campos de la estructura:

- `chars`: Puntero a bloque de memoria donde almacenamos los caracteres de la línea.
- `capacity`: Capacidad del arreglo; nos sirve para cuando tenemos que redimensionar por falta de lugar.
- `used`: Indica qué parte del arreglo está siendo utilizada.

2.2.3. Expandir DString

En los casos que teníamos que agregar caracteres al `DString` y no teníamos capacidad, usamos la función `dstring_extend()` para ampliar la capacidad de la estructura. Esta extensión la hicimos en función del factor `RESIZE_FACTOR`.

2.3. Dictionary

El tipo `Dictionary` no es más que un `CTrie` con algunas funcionalidades agregadas. Creamos el tipo `Dictionary` para agregar una capa más de abstracción, ya que nuestro diccionario cumplía ciertas características adicionales, que terminamos implementando internamente con un `CTrie`. Las funcionalidades principales son:

- `dictionary_load_from_file()`: Se encarga de insertar todas las palabras de un archivo `.txt` que consideramos como diccionario, dentro de la estructura `Dictionary`. Internamente se implemento utilizando la función de `CTrie`, `ctrie_add_string()`.
- `dictionary_largest_prefix()`: Nos devuelve la máxima longitud de prefijo que podemos encontrar desde cierta posición de la línea y con un determinado diccionario. Esta función recorre el `Dictionary` siempre y cuando los caracteres de la línea a parsear coincidan con los caracteres de las palabras cargadas en la estructura. Una vez que encuentra una palabra válida, el algoritmo guarda dicha longitud, y sigue buscando para intentar superar esa longitud encontrada. Funciona como la búsqueda de una palabra en el diccionario, solo que lo marca como un *checkpoint* y sigue buscando.

3. Parseo de archivos

En la sección anterior, mostramos las estructuras de datos que utilizamos y explicamos un poco su funcionamiento. Ahora, mostraremos cómo terminamos relacionando las estructuras y sus funcionalidades, para que podamos llegar a parsear un archivo completo. Podríamos decir que el parseo de un archivo completo se divide en las siguientes etapas:

- Carga del diccionario de palabras válidas.
- Parseo de líneas individuales del archivo.
  - Búsqueda de máximo prefijo.
  - Recuperación de errores.

Durante el parseo, utilizamos la función `get_parse_char()`, que aprovecha lo que mencionamos en la explicación del `DString`: leer del archivo si nunca habíamos leído esa parte de la línea, y leer del propio `DString` cuando ya hayamos leído esa sección de la línea.

3.1. Carga del diccionario

La carga del diccionario la realizamos directamente en la función `main()`, utilizando la función `dictionary_load_from_file()` y el primer archivo pasado como argumento al programa.

3.2. Parseo por líneas

Los punteros de los archivos a parsear y donde guardaremos los resultados del parseo (ambos pasados como argumentos por consola al programa), se pasan como argumentos, junto al diccionario ya cargado, a la función `parse_file()`, la cual internamente llama a la función `parse_line()`. Veamos qué ocurre dentro la última función.

3.2.1. Búsqueda de máximo prefijo

En los casos que `dictionary_largest_prefix()` nos devuelva un valor positivo, significa que hemos encontrado un prefijo desde la posición en la que nos encontramos. Utilizamos la función `dstring_save_segment()` para guardar la palabra de largo máximo en el archivo de resultados. Luego, avanzamos tantas casillas como el largo de palabra encontrada, y continuamos parseando hasta llegar al final de la línea.

3.2.2. Recuperación de errores

En los casos que `dictionary_largest_prefix()` nos devuelva 0, significa que no se ha podido alcanzar ninguna palabra válida desde dicha posición, por lo que almacenamos dicho caracter como un error y avanzamos una casilla, para seguir parseando el resto de la línea. Los errores los fuimos guardando en un `DString`, para que luego de terminar de parsear la línea, los podamos guardar en el archivo de resultados. Además, para mejorar la lectura de los errores, fuimos agregando un espacio entre cada bloque de errores: cuando encontrábamos una palabra valida, los siguientes errores que encontrábamos consecutivamente los agrupábamos.

En primera instancia, tratamos de ir almacenando los errores en una `Queue` implementada con `GList`. Terminamos descartando dicha idea ya que en los casos que se utilizaba una gran cantidad de veces la función `queue_enqueue()` para encolar un error, se realizaba una elevada cantidad de llamadas al sistema por parte de `malloc()`, lo que provocaba que el programa demore un tiempo considerable.

4. Organización de archivos y carpetas

Ahora daremos una rápida explicación de cómo organizamos los directorios y archivos del programa:

- `src/`: Se encuentran los archivos relacionados al `Dictionary` y las funciones de parseo. También encontramos el archivo `main` del programa.
- `structures/`: Archivos relacionados a las estructuras de datos `CTrie` y `DString`.
- `testdata/`: Archivos de ejemplo para parsear.
- `dictionaries/`: Diccionarios de ejemplo para el parseo de las líneas.
- `test/`: Archivos relacionados al testeo de las funciones del programa.

5. Compilación y ejecución

La compilación y ejecución del programa se encuentran detalladas en el `README` proporcionado en la carpeta del trabajo práctico. De igual manera, haremos una explicación a continuación.

5.1. Compilación de archivos

Primero veamos la compilación, para la cual se realizó un archivo `makefile` para facilitarla. Las principales reglas son:

- `make / make all`: Compila el parser y elimina los archivos objeto.
- `make parser`: Compila únicamente el parser.
- `make test`: Compila únicamente los tests.
- `make clean`: Elimina todos los archivos objeto.

El comando `make` se tiene que ejecutar en el directorio raíz del proyecto.

5.2. Ejecución del parser y los tests

Ahora veamos la ejecución del parser y de los tests. Para la ejecución del parser, tenemos que ubicarnos en el directorio del proyecto y ejecutar el comando `./parser`, al cual le deberemos pasar tres argumentos:

- **Diccionario**: Diccionario en formato `.txt` de palabras válidas para parsear el archivo.

- **Archivo a parsear**: Archivo en formato `.txt` que queremos parsear.
- **Archivo de resultados**: Archivo en formato `.txt` en donde guardaremos los resultados del parseo.

Los tres archivos pasados como argumentos, deben incluir la ruta relativa al directorio raíz del proyecto, que es el lugar donde se encuentra el ejecutable del parser.

Un ejemplo de ejecución sería:

```
./parser dictionaries/small_dictionary.txt testdata/prueba.txt result.txt
```

Por otro lado, para ejecutar los tests, simplemente debemos ejecutar el comando `./test` en el directorio raíz del proyecto.

6. Conclusión

Podemos decir que obtuvimos un resultado satisfactorio con el trabajo práctico. Reutilizamos cosas aprendidas en la materia, como el `DString`, y conocimos estructuras nuevas como el `Trie` o `CTrie`. También pudimos experimentar con diferentes estructuras, y los beneficios que tienen unas sobre las otras; como en el caso que nos decantamos por el `DString` en lugar de la `Queue`.

También podemos ver que se aprendieron muchas cosas durante la realización del trabajo practico. Como mencionamos antes, tuvimos que probar entre distintas alternativas y, bajo nuestro criterio, determinar cual era la mas óptima para la situación que se nos presentaba. Profundizamos en el uso de ciertas herramientas que fueron de mucha ayuda, y se pudo mejorar en ciertas áreas, desde la planificación y estructuración del proyecto antes de comenzar a programar, hasta la habilidad de encontrarse frente a un problema e idear una forma de resolverlo de manera óptima con conceptos ya conocidos o aprendidos durante el transcurso del proyecto.