Trabajo Práctico
Union-Find de la Copa América

Estructuras de Datos
Tecnicatura en Programación Informática
Universidad Nacional de Quilmes

1. Introducción

Existen diversos algoritmos que requieren agrupar elementos de un conjunto según un determinado criterio, en conjuntos que son disjuntos entre sí[1], para luego ir combinando dichos conjuntos conforme progresa el algoritmo. Una estructura de datos apropiada para este tipo de problemas es la denominada *conjunto union-find*[2], que abreviaremos *UFSet* de ahora en más.

Un *UFSet* implementa de manera eficiente las operaciones *find* (encontrar el conjunto al que pertence un elemento dado), y *union* (combinar dos conjuntos mediante su unión). En este trabajo práctico pediremos implementar una versión imperativa de *UFSet*, utilizando el lenguaje `C/C++` como se utiliza en la materia.

2. Motivación

Esta sección explica un ejemplo trivial para ilustrar el tipo de problema que describimos en la introducción.

Problema: Dado un conjunto S de equipos de fútbol que juegan la *Copa América*, agruparlos según el grupo que les toca en la copa. Por ejemplo, sea

$$S = \{argentina, chile, venezuela, colombia, paraguay, mexico, uruguay\}$$

Si sabemos que Paraguay y Colombia juegan en el grupo A, Venezuela, Uruguay y México en el C, y Argentina y Chile en el D, el conjunto de grupos resultante será entonces:

$$\{\{paraguay, colombia\}, \{venezuela, mexico, uruguay\}, \{argentina, chile\}\}$$

Notar también que los grupos son *disjuntos*, en este caso porque un elemento pertence a lo sumo a un grupo.

Solución: Podemos solucionar el problema mediante este algoritmo:

1. Calcular todos los pares de elementos (x, y) donde x comparte grupo con y.
2. *Crear* un conjunto P que contenga a cada elemento del conjunto S en cuestión como un conjunto unitario. Matemáticamente: $P = \{\{i\} : i \in S\}$.
3. Para todo par (x, y) de elementos del paso (1)
   a) *Encontrar* el conjunto C_x al que pertenece x

[1] Un conjunto es disjunto en relación a otro si no comparten ningún elemento
[2] Conocidas como *disjoint-set data structure* o *union–find data structure*. Para más información consultar el link a estructuras de datos para conjuntos disjuntos de Wikipedia.

b) *Encontrar* el conjunto C_y al que pertenece y

c) En P, reemplazar a C_x y C_y por $C_{xy} = C_x \bigcup C_y$, es decir, la *unión* de C_x y C_y

Al finalizar el algoritmo, P contendrá a los conjuntos buscados.

Apliquemos el algoritmo a nuestro ejemplo:

1. Los pares (x, y) donde x comparte grupo con y son:
   $\{(paraguay, colombia), (venezuela, mexico), (argentina, chile),$
   $(venezuela, uruguay), (uruguay, mexico), (colombia, paraguay),$
   $(mexico, venezuela), (chile, argentina), (uruguay, venezuela), (mexico, uruguay)\}$

   Observar que como la relación es simétrica, si $(paraguay, colombia)$ es un par válido, también lo es $(colombia, paraguay)$, y según la definición que dimos para este punto, debemos incluirlo, pese a que, como veremos, no va a influir en el resultado final.

2. Nuestros conjuntos iniciales están dados por
   $P = \{\{argentina\}, \{chile\}, \{venezuela\}, \{colombia\}, \{paraguay\}, \{mexico\}, \{uruguay\}\}$

3. Para el último paso, consideremos los pares obtenidos en el paso (1).

   a) Al procesar $(paraguay, colombia)$, queda
   $P = \{\{paraguay, colombia\}, \{argentina\}, \{venezuela\}, \{chile\}, \{mexico\}, \{uruguay\}\}$.

   b) Al procesar $(venezuela, mexico)$, queda
   $P = \{\{paraguay, colombia\}, \{venezuela, mexico\}, \{argentina\}, \{chile\}, \{uruguay\}\}$.

   c) Luego de procesar $(venezuela, uruguay)$, tenemos
   $P = \{\{paraguay, colombia\}, \{venezuela, mexico, uruguay\}, \{argentina\}, \{chile\}\}$.

   d) Finalmente, al procesar $(argentina, chile)$, tenemos
   $P = \{\{paraguay, colombia\}, \{venezuela, mexico, uruguay\}, \{argentina, chile\}\}$.

   Los pares siguientes son de elementos que ya están en el mismo grupo, y los ignoraremos en la presentación del ejemplo ya que no modifican el resultado.

Es importante entender que el algoritmo sirve para casos en donde incluso no sabemos cuántos elementos potencialmente podemos llegar a comparar, y, sobre todo, cuántos conjuntos disjuntos se van a formar, ni cuántos elementos va a tener cada uno. Con esto queremos decir que el ejemplo de la Copa América es trivial, dado que sabemos que juegan exactamente 16 equipos, en grupos de 4, y su ordenamiento en grupos disjuntos resulta simple aún sin aplicar un algoritmo como este. No obstante, nos alcanza con este problema para probar una implementación de *UFSet*.

Nota: Si ya cursaron Matemática 1, este algoritmo construye una partición de un conjunto[3] dada una relación de equivalencia[4].

Por último, aclaramos que el algoritmo usa como estructura auxiliar un *UFSet* para operar de forma eficiente, pero los *UFSet* en realidad no conocen qué relación hay entre los elementos. Si a un *UFSet* le pedimos encontrar y unir dos equipos que en realidad no juegan en el mismo grupo, lo hará de todas formas, dado que es el algoritmo el que indica al *UFSet* qué debe unir en cada paso. Por eso los *UFSet* son una estructura muy simple: sólo buscan y unen los conjuntos que digan sus usuarios.

[3] Link a partición de un conjunto en Wikipedia.
[4] Link a relación de equivalencia en Wikipedia.

3. Descripción del Trabajo

El algoritmo antes descrito requiere una representación de conjuntos que implemente eficientemente las operaciones: (1) *crear* un conjunto con un único elemento, (2) *encontrar* el conjunto que contiene a un elemento dado, y (3), *combinar mediante la unión* 2 conjuntos disjuntos.

El trabajo consiste en implementar en el lenguage `C/C++` la estructura de datos *UFSet*, que implemente de manera eficiente las operaciones combinar (union) y encontrar (find), y las optimizaciones detalladas en las secciones 4.3 y 4.4.

Este enunciado contiene una guía paso a paso para que esto resulte simple. Este TP está pensado para ser resuelto de forma gradual, y primero haremos una implementación, para luego darnos cuenta que no es la mejor, y pasaremos a optimizarla. Eso nos permitirá probar primero si entendimos de qué trata.

En resumen, se pide hacer los siguientes pasos:

(a) **Definir una representación para el tipo *UFSet*:** este paso es *crucial*, ya que la representación y los invariantes que se elija determinará cómo implementar las operaciones de la estructura, así como las funciones auxiliares que podamos llegar a pedir para definirlas.

(b) **Implementar las operaciones *create*, *find* y *union*:** una vez definido el tipo *UFSet*, implementaremos *find* y *union*, según su descripción en la sección 4.2.

(c) **Implementar la mejora de unión por rango:** consiste en modificar la operación *union*, tal como explicamos en la sección 4.3. Para esto, será necesario también modificar el tipo *UFSet* para incluir información de rango en los nodos del árbol.

(d) **Implementar la mejora de compresión de camino:** consiste modificar la operación *find*, tal como describimos en la sección 4.4.

(e) **Probar su correcto funcionamiento:** modelaremos el ejemplo de la Copa América siguiendo los pasos de la sección 4.6, para poder correr pruebas y verificar el correcto funcionamiento de la implementación que realicen.

4. Estructura de Datos *UFSet*

4.1. Descripción del tipo abstracto

Vamos a representar un *UFSet* como un elemento "distinguido", cualquiera, de esa misma estructura. En el ejemplo de los grupos de la Copa América, como Argentina juega en el grupo D, es igualmente válido referirse a este grupo como "grupo D" o el "grupo de Argentina", ya que ningún país está en más de un grupo, y por lo tanto al hablar de Argentina está perfectamente claro que nos referimos al grupo D. Entonces, si nos preguntan en qué grupo está Chile, podemos decir "en el grupo de Argentina", y va a ser lo mismo que decir "grupo D". Y lo mismo podemos hacer con cualquier conjunto, tomamos a un elemento que pasa a ser el representante de todos, y así, el conjunto donde tenemos un elemento "x" pasa a ser sinónimo de "el de x". Notar que es igual de válido tomar a cualquier elemento de un conjunto como dicho representante.

Entonces, la idea clave para esta estructura es: *el tipo que utilizaremos para modelar un UFSet será el mismo que usaremos para representar sus elementos.* **No habrá distinción entre un elemento de un *UFSet* y un *UFSet*.**

Con esto no nos estamos refiriendo a la representación interna del tipo abstracto, sino al tipo abstracto en sí. El TAD `UFSet` y un elemento de ese TAD, serán para nosotros indistinguibles. Es fundamental comprender esta dualidad para finalizar exitosamente este trabajo.

Esto mismo lo podemos notar en las operaciones más importantes de un *UFSet*, que son las siguientes:

UFSet **find**(*UFSet e*)	— retorna el conjunto que contiene al elemento e
UFSet **union**(*UFSet* set_1, *UFSet* set_2)	— retorna la unión de los conjuntos set_1 y set_2

Observar que la operación *find* pone de manifiesto la dualidad elemento-conjunto: como representamos a un *UFSet* mediante un elemento distinguido, tanto el elemento e como el conjunto a retornar corresponden al tipo abstracto *UFSet*. Sin embargo, e puede ser cualquier elemento del conjunto (al verlo como elemento), mientras que el resultado de $find(e)$ siempre será el elemento elegido como distinguido (al verlo como elemento).

4.2. Primera implementación

El tipo de representación del TAD *UFSet* estará dado por la forma en que implementemos las operaciones fundamentales de esta estructura de datos. Haremos entonces una primera descripción de una implementación de sus operaciones, para que se pueda deducir cómo expresar en lenguaje `C/C++` el tipo de representación esperado.

4.2.1. find

Como dijimos, la operación *find* consistirá en encontrar el elemento distinguido de un conjunto a partir de otro elemento recibido como parámetro. Por lo tanto, deberíamos ser capaces de "navegar" desde todo elemento de un conjunto hasta su elemento distinguido. Una manera de implementar esto es pensar en un *UFSet* como un tipo particular de árbol donde:

(a) Sus nodos son los elementos del conjunto, de tipo *UFSet*

(b) Su raíz es el elemento distinguido del conjunto

(c) Todo nodo tiene una referencia a su padre, de modo que desde cualquier nodo del árbol podemos alcanzar su raíz.

Observar que en este "árbol", a diferencia de otros árboles que hemos trabajado, un nodo NO conoce a sus hijos, sino a su *padre*. Es, por lo tanto, imposible de representar en Haskell, ya que requiere características imperativas. Observar también que si le preguntamos a un elemento distinguido por su conjunto, deberá devolverse a sí mismo como respuesta, por lo que podría decirse que "es su propio padre"...

De esta manera, la implementación de *find* consistirá en, dado un nodo de tipo *UFSet*, que es un elemento del conjunto, retornar la raíz del árbol al que pertence el nodo. Está operación será `O(n)` en peor caso, dado que podemos estar recorriendo toda una rama, desde una hoja hasta la raíz del árbol, hasta devolver al representante del conjunto.

4.2.2. union

A su vez, la operación *union* deberá, dados dos elementos de un conjunto, encontrar la raíz de los árboles a los que pertencen ambos nodos, y hacer que una de las dos raíces sea el padre de la otra.

La figura 1 ilustra el proceso de unión de conjuntos. Arrancamos con 4 conjuntos unitarios: $ce1$, $ce2$, $ce3$ y $ce4$. Los nodos grises representan al nodo distinguido de un conjunto. Como cada uno es unitario, su único nodo debe ser el elemento distinguido. Luego unimos $ce1$ con $ce2$, para obtener $ce12$. Notar cómo, en efecto, estamos haciendo que una de las raíces (en este caso $ce1$) pase a ser padre de la otra. La raíz que se mantiene como padre será el elemento distinguido de la unión. Hacemos lo mismo con $ce3$ y $ce4$, para obtener $ce34$. Finalmente, unimos $ce12$ y $ce34$ mediante el mismo procedimiento, esto es, transformando a uno de los dos elementos distinguidos en hijo del otro. En la figura, el elemento distinguido de $ce34$ pasa a ser el hijo del de $ce12$. El elemento distinguido de este nuevo conjunto será, por lo tanto, la raíz de $ce12$.

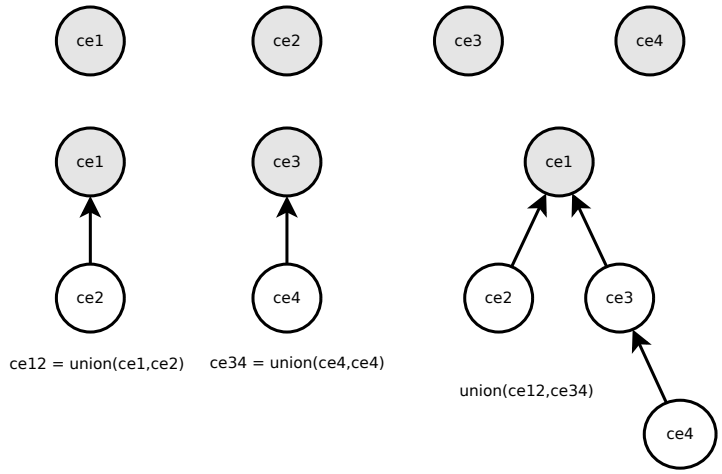

Figura 1: Unión de *UFSet*

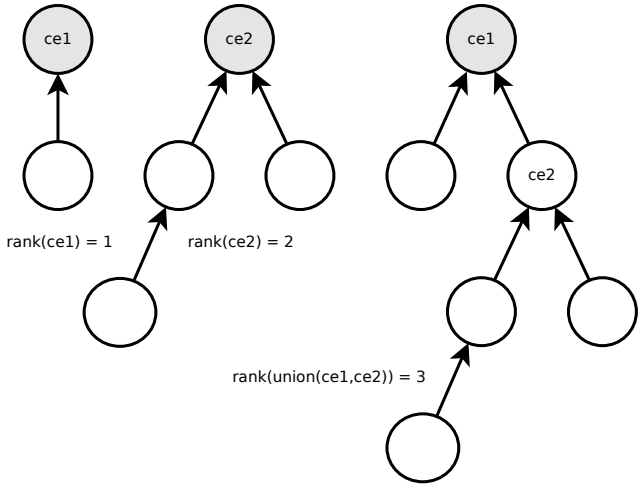

Figura 2: Mala elección de la raíz al unir $ce1$ y $ce2$

4.3. Optimización 1: unión por ranking

Al unir dos *UFSet*, estamos eligiendo arbitrariamente al primer nodo distinguido como raíz del segundo. Veamos por qué esto es una mala idea.

Definición: definimos al *rango* de un *UFSet* como la altura del árbol que lo representa, es decir, la distancia máxima desde su raíz (el nodo distinguido) hasta alguna de sus hojas.

La implementación de *find*, en el peor caso, deberá recorrer un camino con largo igual al rango del conjunto (eso sucederá cuando reciba por parámetro sea una hoja). Por este motivo, al unir dos conjuntos es deseable minimizar el rango. La figura 2 muestra lo que sucede cuando el rango del primer conjunto es menor que el del segundo. Al agregar $ce2$, el conjunto de mayor rango, como hijo de $ce1$; terminamos con un nuevo conjunto cuyo rango es 3.

¡Pero podemos hacerlo mejor! La figura 3 muestra el resultado de poner a $ce1$, el conjunto de menor rango, como hijo $ce2$. El nuevo resultado tiene rango 2. En general, al unir dos conjuntos,

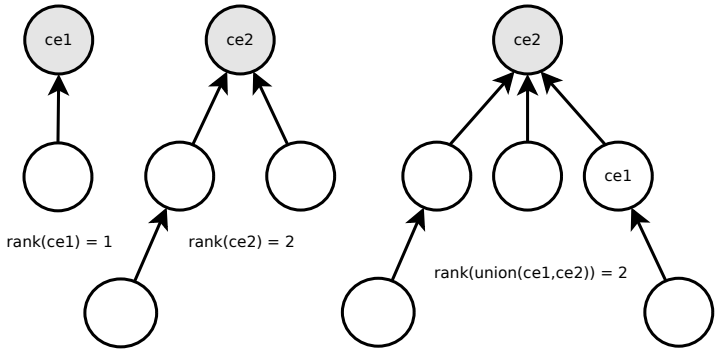

Figura 3: Buena elección de la raíz al unir $ce1$ y $ce2$

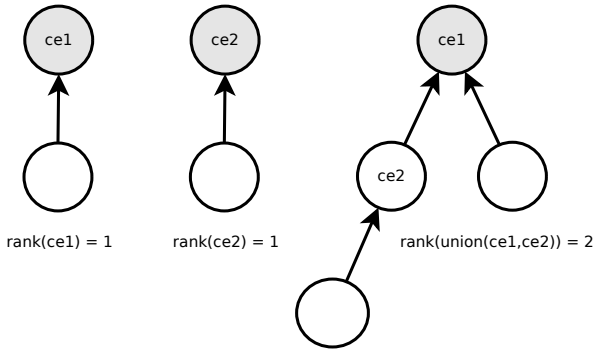

Figura 4: Al unir dos conjuntos con igual rango, el rango resultante se incrementa en uno

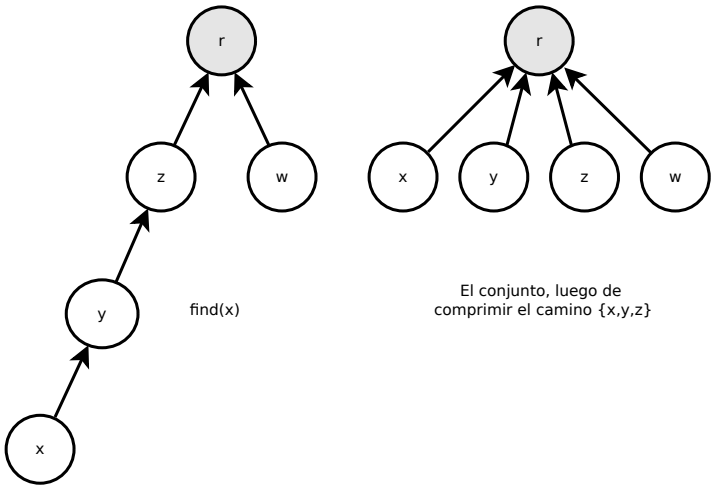

Figura 5: Compresión del camino $\{x, y, z\}$

la estrategia óptima consiste en **agregar al árbol de menor rango como hijo del de mayor rango**. Siguiendo esta regla, *la única manera de que el rango crezca es unir conjuntos con rangos iguales*, en cuyo caso se incrementará en una unidad. Podemos ver este proceso en la figura 4.

Entonces, por la dualidad elemento-conjunto de *UFSet*, para implementar esta optimización debe llevarse cuenta del rango en cada nodo dentro del mismo (actualizando los invariantes de representación), y utilizarlo adecuadamente al hacer la unión (recordando recalcularlos de ser necesario).

4.4. Optimización 2: compresión de camino

Nuestra última optimización será sobre la operación *find*. Al finalizar el recorrido desde un nodo hasta su raíz, conocemos dos cosas: (1) el nodo raíz, y (2), los nodos que forman parte del camino hasta ese nodo raiz. Con esta información, es posible *comprimir* dicho camino: sólo necesitamos que el padre de cada nodo visitado (excepto la raíz) pase a ser la raíz encontrada.

La figura 5 muestra lo que sucede al hacer *find*(x): todos los nodos del camino $\{x, y, z\}$ hasta la raíz r pasan a ser hijos directos de r. La próxima vez que ejecutemos *find*(x) (ó y, ó z) recorreremos un camino de largo 1.

Para implementar esta optimización, cada vez que se realice un *find* deben reasignarse los nodos padre (recordando recalcular el rango de forma adecuada).

4.5. Material de Soporte

Para guiar la implementación de *UFSet* y además poder probarla, se provee el siguiente material, accesible como un archivo zip en el aula virtual de la materia.

(a) `UFSet.h`: Contiene interfaz de *UFSet*.

(b) `UFSet.cpp`: Template donde implementar la interfase declarada en `ufset.h` y las optimizaciones correspondientes.

(c) `Test.cpp` y `Test.h`: Módulo que sirve para probar de forma completa que la implementación de *UFSet* funciona para ciertos casos.

(d) `main.cpp`: Módulo para probar la implementación. Se trata del problema descripto en la sección 2, esto es, agrupar equipos que juegan en el mismo grupo. Se provee un test completo

que utiliza el módulo `Test`, pero se puede (y se debería) plantear otros casos para probar el código en forma completa.

4.6. Pruebas

Una vez completada alguna versión de `UFSet.cpp`, debe probarse la implementación compilando el proyecto y ejecutando el material de soporte provisto para el trabajo.

El trabajo **debe** funcionar correctamente. Si las pruebas indican que hay algún error en la implementación, no dejar de consultar. Nosotros vamos a utilizar estas pruebas (entre otras) para corregir tu trabajo.

4.7. Forma de Entrega

La entrega consistirá en mandar a la lista de correos de la materia el archivo `UFSet.cpp` y cualquier otro que se modifique o cree para realizar testeos en forma de archivo *zip*, antes de las fechas establecidas (según se trate de la entrega original, o de la entrega recuperatoria). El envío se realiza por mail a la lista de docentes de la materia[5]. Tanto el nombre del zip como el *subject* del mail deberán ser: **ED2023s2-TPFinal-<*Apellido*>-<*Nombre*>**. Ejemplo: "`ED2016-TPFinal-Perez-Romualdo.zip`". Entregas que no respeten esta consiga con exactitud se considerarán como NO ENVIADAS[6].

Dado que se trata de una evaluación, habrá una entrega original y una entrega recuperatoria.

- La fecha máxima para la entrega original es el 23/11, 23:59hs.
- La fecha máxima para la entrega recuperatoria es el 5/11, 23:59hs.
- La fecha de la defensa de aquellos trabajos aprobados en alguna de las dos instancias es el 12/11, a las 16hs.

Todo trabajo que haya sido entregado antes de la fecha máxima de entrega original será corregido y en caso de requerirse, podrá enviarse una nueva entrega antes de la fecha máxima de la entrega recuperatoria. Las entregas realizadas entre ambas fechas máximas serán corregidas, pero NO podrán reenviarse. El máximo de entregas por trabajo será 2, siempre que sea dentro de las fechas antedichas.

Si un trabajo fuese calificado con nota de aprobación, pero no aprobase la defensa correspondiente, dicha nota no será tenida en cuenta, y la materia NO se considerará aprobada.

4.8. Forma de Evaluación

De aquellas entregas que hayan sido realizadas en tiempo y forma con las implementaciones solicitadas, según las fechas descriptas, evaluaremos fundamentalmente los siguientes aspectos:

(a) El proyecto debe compilar bien. Si esto no sucede, el trabajo se considera desaprobado, sin posibilidades de ser defendido. Consejo: **compilá todo el tiempo tu código**. No avances con otras cuestiones hasta que no hayas logrado esto, y pedí ayuda en lo que respecta al lenguaje y las herramientas que usamos.

(b) La implementación correcta de *UFSet*. Recordá que podés pedir ayuda tanto como sea necesario. Pero dicho esto, la implementación debe **ejecutar correctamente**, y, de no hacerlo, el trabajo se considera desaprobado.

[5] tpi-doc-ed@listas.unq.edu.ar

[6] Observar que el Apellido y Nombre DEBEN comenzar con mayúsculas, y aparecer en ese orden. Observar también que el nombre del archivo DEBE repetirse en el tema del mail. Y finalmente observar que el formato del archivo debe ser `.zip` (no `.rar` u otro formato de compresión), y DEBE contener al menos al archivo `UFSet.cpp` y cualquier otro que se haya modificado, pero NO los que NO se modificaron.

(c) El estilo de la materia. Que tu implementación compile y funcione correctamente no es garantía de que hayas utilizado los conceptos vistos en la materia. Es muy importante que, por ejemplo, respetes barreras de abstracción, indiques los invariantes de representación, precondiciones, etc. Un trabajo con graves faltas en este sentido no estará aprobado con buena nota, e incluso puede llegar a estar desaprobado, si es que no cumple en ninguna de las consignas vistas durante la materia.

Como se indicó, adicionalmente deberán defender su trabajo para demostrar que entienden lo que entregaron. Está bien buscar ideas en internet y/o consultar compañeros, pero deben *demostrar* que entienden lo que entregan. Por eso, luego de la entrega se tomará un ejercicio adicional, trivial, para demostrar esta comprensión. Este ejercicio adicional es tan importante como el TP. Si este ejercicio no es realizado correctamente presumiremos que el trabajo entregado no se comprende, y la nota que hayamos asignado NO será considerada, desaprobando el trabajo. Entonces, deben procurar entender el código, ya sean el único autor, o hayan recibido ayuda (idealmente, deberían intentar hacerlo solos, pues es un buen entrenamiento para el trabajo más exigente en un ambiente laboral).

5. Conclusiones

Este trabajo es un desafío. Resolverlo requiere comprender un enunciado largo, un mínimo dominio de los temas que vemos en la segunda parte de la materia, incluyendo algo de `C/C++` (un lenguaje no necesariamente sencillo de aprender), y finalmente, implementar y optimizar una estructura de datos nueva, no vista hasta el momento. Nuevamente, nos gustaría hacer hincapié en lo siguiente: **no dudes en recurrir a los docentes si precisás ayuda**.

Quizás la enseñanza más importante de este trabajo sea que existen otras estructuras de datos además de las vistas en las teóricas. Podés ver a los *UFSet* como una herramienta más en tu "caja de herramientas" (algoritmos y estructuras de datos que podés aplicar al momento de resolver un problema). Si bien agrupar equipos según qué grupo les tóco jugar no se vé como un problema particularmente complejo, la idea general (formar grupos mediante la combinación de grupos más pequeños) es extremadamente útil, y existe una buena cantidad de algoritmos cuya implementación eficiente requiere utilizar *UFSet*. Un ejemplo de tales algoritmos es el algoritmo de *Kruskal*[7], utilizado para diseñar redes con costo mínimo, y generación automática de laberintos[8]. No hemos utilizado dichos algoritmos como ejemplo ya que exceden el alcance de esta materia, pero si te interesa cursar la Licenciatura en Desarrollo de Software, recomendamos que mires, por lo menos de forma superficial, artículos relacionados con esos temas.

[7] Link a algorimo de Kruskal en Wikipedia.
[8] Link a algoritmos de generación de laberintos en Wikipedia.