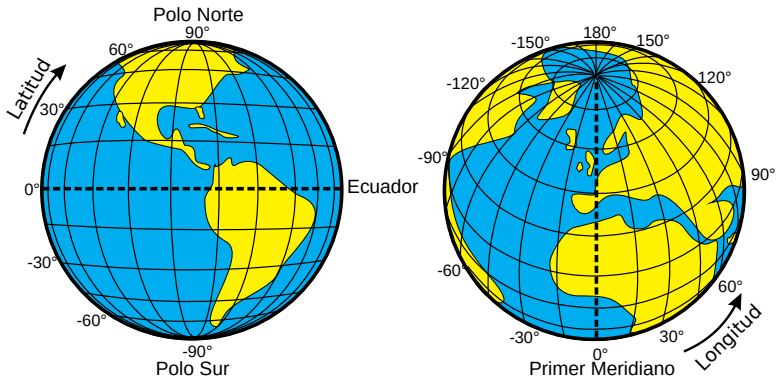
Trabajo Práctico 2

Distancia Geográfica

Programación en C

Tecnología Digital II



1. Introducción

El objetivo de este trabajo práctico es implementar una estructura de datos que nos permita almacenar una lista de ciudades con sus coordenadas. Esta estructura nos va a permitir obtener la distancia total de recorrer todas las ciudades una por una. Además, queremos implementar funciones que nos permitan mejorar este camino, ya sea modificando la posición de las ciudades arbitrariamente o aplicando una heuristica.

> El trabajo práctico debe realizarse en grupos de tres personas. Tienen tres semanas para realizar la totalidad de los ejercicios. **La fecha de entrega límite es el 1 de Junio hasta las 23:59; y la de re-entrega es el 22 de Junio hasta las 23:59**
> Se solicita no realizar consultas del trabajo práctico por canales públicos. Limitar las preguntas al foro privado creado para tal fin.

2. Tipos de datos: `Path`

A partir de las siguientes estructuras se define un `path`:

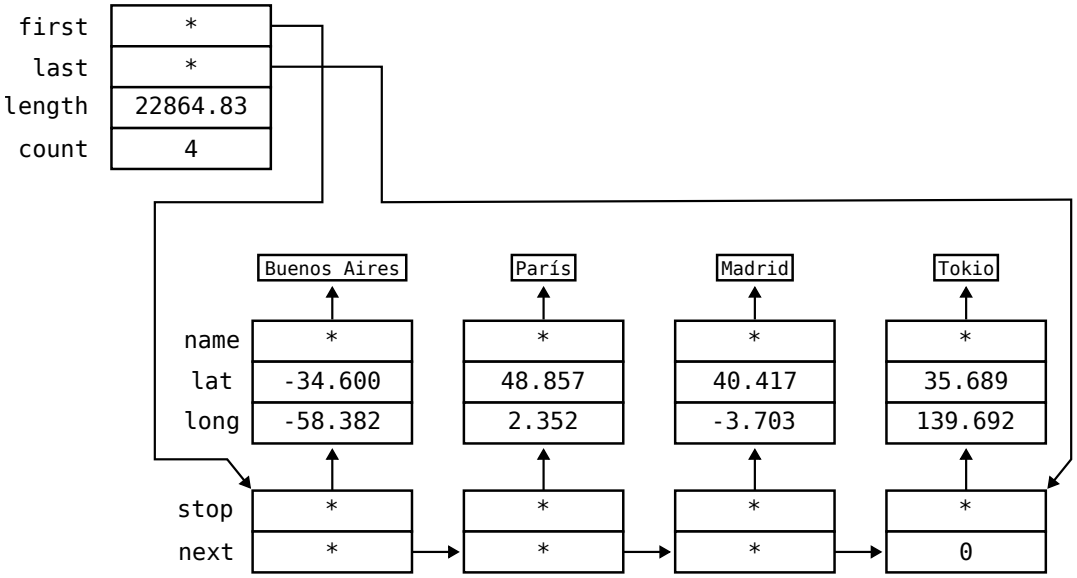
```
struct city {
    char* name;
    float latitude;
    float longitude;
};

struct node {
    struct city* stop;
    struct node* next;
};
```

```
struct path {
    struct node* first;
    struct node* last;
    float length;
    int count;
};
```

La estructura `path`, contiene un par de punteros al primero y último nodo de una lista construida con la estructura `node`. Además esta estructura cuenta con los campos `length` y `count`, correspondientes a la cantidad de kilómetros del camino de todas las ciudades y a la cantidad de ciudades

respectivamente. Los nodos construyen una lista simplemente enlazada con su último nodo apuntando a *null*. Los nodos contienen solo un puntero denominado `stop` de tipo `city`. Las estructura `city` por su parte cuenta con solo tres campos, correspondientes al nombre de la ciudad y sus coordenadas. A continuación se ilustra un ejemplo de la estructura:



En el ejemplo se identifica un camino entre cuatro ciudades, que comienza en Buenos Aires y termina en Tokio. Primero pasa por Paris, luego por Madrid y por último se llega a Tokio.

3. Enunciado

Ejercicio 1

Implementar las siguientes funciones sobre *strings*.

- `int strLen(char* src)`
  Calcula la longitud de una *string* pasada por parámetro, indicando como respuesta la cantidad de caracteres de esta.
- `char* strDup(char* src)`
  Duplica un *string*. Debe contar la cantidad de caracteres totales de *src* y solicitar la memoria equivalente. Luego, debe copiar todos los caracteres a esta nueva área de memoria, agregando como último byte un *null*. Además, como valor de retorno se debe retornar el puntero al nuevo *string*. Esta función NO debe borrar la *string* original.
- `int strCmp(char* a, char* b)`
  Compara dos *strings* en orden lexicográfico[1]. Debe retornar:
  - 0 si son iguales
  - 1 si `s1` $<$ `s2`
  - -1 si `s2` $<$ `s1`

  Ejemplos:
  `strCmp("ala","perro")` $\rightarrow$ `1`
  `strCmp("casa","cal")` $\rightarrow$ `-1`
  `strCmp("topo","top")` $\rightarrow$ `-1`
  `strCmp("pato","pato")` $\rightarrow$ `0`

[1] `https://es.wikipedia.org/wiki/Orden_lexicografico`

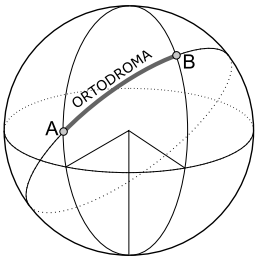
Ejercicio 2

Implementar las siguientes funciones sobre el tipo de datos `path`.

- `struct path* pathNew()`
  Construye un `path` vacio. Todos sus punteros son *null*, la cantidad de ciudades es cero y la distancia es también cero.
- `void pathAddFirst(struct path* p, char* name, float latitude, float longitude)`
  Agrega una nueva ciudad en el primer lugar del camino. Para esto debe crear la estructura de la ciudad y duplicar la string con el nombre. Además debe actualizar la estructura de `path` según corresponda, tanto aumentando la cantidad de ciudades, como modificando la longitud del camino.
- `void pathAddLast(struct path* p, char* name, float latitude, float longitude)`
  Equivalente a `pathAddFirst` pero agregando la lista al final del camino.
- `struct path* pathDuplicate(struct path* p)`
  Duplica un `path`, para esto debe copiar toda la memoria almacenada, haciendo una copia tanto de la estructura, como los nodos y las ciudades. Incluso copiando también los nombres de las ciudades. El resultado no puede compartir ningún puntero con la estructura original.
- `void pathSwapStops(struct path* p, char* name1, char* name2)`
  Intercambia dos ciudades del camino. Toma como parámetro el nombre de ambas ciudades.
- `void pathRemoveCity(struct path* p, char* name)`
  Borrra una ciudad del camino a partir de su nombre.
- `void pathDelete(struct path* p)`
  Borra toda la estructura.
- `void pathPrint(struct path* p)`
  Imprime la estructura en pantalla.

Además, se recomienda implementar las siguientes funciones auxiliares:

- `float calculateLength(struct node* n)`
  Dado un puntero al primer nodo de un camino, calcula la longitud total del mismo. Tener en cuenta que si el camino tiene una sola ciudad, el resultado deberá ser cero, al igual que si no tiene ciudades que recorrer.
- `struct node* findNodeCity(struct node* n, char* name)`
  Dado un puntero al primer nodo de un camino y el nombre de una ciudad, encuentra el nodo que corresponde a la ciudad.
- `float distance(struct city* c1, struct city* c2)`
  Calcula la distancia entre dos ciudades utilizando la formula de Haversine. Esta función simplifica la distancia sobre la tierra, considerando que la misma es una esfera perfecta de radio 6371 km.



Para facilitar el desarrollo, las funciones `distance` y `pathPrint` vienen dadas.

Ejercicio 3

A continuación, se enumera un conjunto mínimo de casos de test que deben implementar dentro del archivo `main`:

- `strLen`
  1. String vacío.
  2. String de un carácter.
  3. String que incluya todos los caracteres alfanuméricos.
- `strDup`
  1. String vacío.
  2. String de un carácter.
  3. String que incluya todos los caracteres alfanuméricos.
- `strCmp`
  1. Dos string vacíos.
  2. Dos string de un carácter.
  3. Strings iguales hasta un carácter (hacer `cmpStr(s1,s2)` y `cmpStr(s2,s1)`).
  4. Dos strings diferentes (hacer `cmpStr(s1,s2)` y `cmpStr(s2,s1)`).
- `pathAddFirst`
  1. Agregar una ciudad a un `path` vacío.
  2. Agregar una ciudad a un `path` con una sola ciudad.
  3. Agregar una ciudad a un `path` con más de una ciudad
- `pathAddLast`
  1. Agregar una ciudad a un `path` vacío.
  2. Agregar una ciudad a un `path` con una sola ciudad.
  3. Agregar una ciudad a un `path` con más de una ciudad.
- `pathDuplicate`
  1. Duplicar un `path` vacio.
  2. Duplicar un `path` con una sola ciudad.
  3. Duplicar un `path` con más de una ciudad.
- `pathSwapStops`
  1. Intercambiar dos ciudades con el mismo nombre. No debería hacer nada, pero no debe fallar.
  2. Intercambiar dos ciudades, de un camino de solo dos ciudades.
  3. Intercambiar dos ciudades, tales que sean la primera y la última del camino.
  4. Intercambiar dos ciudades cualesquiera de un camino de más de 5 ciudades.
- `pathRemoveCity`
  1. Borrar la primer ciudad de un camino de cuatro ciudades.
  2. Borrar la última ciudad de un camino de cuatro ciudades.
  3. Borrar la una ciudad intermedia de un camino de cuatro ciudades.
  4. Borrar la una ciudad de un camino de una sola ciudad.
- `pathDelete`
  1. Borrar un `path` sin ciudades.

2. Borrar un `path` con una sola ciudad.
3. Borrar un `path` con más de una ciudad.

Se recomienda agregar todos los casos de test que gusten, teniendo en cuenta casos borde de borrado y escritura de datos.

Ejercicio 4

Cuando la cantidad de ciudades a recorrer es muy grande, resulta impracticable obtener el camino óptimo. En estos casos se utilizan heurísticas. Este tipo de algoritmos buscan encontrar una muy buena solución que no es necesariamente la mejor.

Se pide entonces implementa alguna de las siguientes heurísticas:

- **Primer más cercano**
  A partir de una ciudad inicial, busca la siguiente ciudad más cercana y coloca está como segunda ciudad del camino. Hace esto repetidas veces para ir encontrando las siguientes ciudades. Termina cuando completa el camino con todas las ciudades.
- **Intercambio de peores**
  Busca el par de ciudades del camino que tengan la mayor distancia entre sí. Aleatoriamente intercambia este par de ciudades con cualquier otra dentro del camino. Si la distancia total mejora, entonces realiza el cambio, si la distancia no mejora, entonces vuelve a intentar con otro par de ciudades aleatoriamente. Termina después de una determinada cantidad de iteraciones.
- **Ordenar coordenadas**
  Ordena las ciudades por sus coordenadas en longitudes. Partiendo de una ciudad inicial, ordena el resto de las ciudades hasta llegar a la última ciudad del recorrido. Es decir, ordena las ciudades teniendo en cuenta si el camino se realiza de este a oeste o a la inversa.

La función a implementar es `struct path* applyHeuristic(struct path* p)`. Tener en cuenta que debe retornar una copia de toda la estructura, tal que las ciudades estén ordenadas según alguno de los criterios anteriores.

Ejercicio 5 (optativo)

Si después de implementar una de las heurísticas sugeridas, se les ocurre que pueden mejorar aún más el camino resultante. Los invitamos a intentarlo y competir entre ustedes para ver quien obtiene el mejor camino.

El equipo que obtenga el mejor camino, recibirá 10 puntos extras en el presente trabajo. Los invitamos a probar todos los algoritmos para buscar la mejor solución.

Utilizando el archivo `cities.csv` como la lista de ciudades a recorrer, se pide armar armar el mejor recorrido yendo desde Buenos Aires a Paris pasando por todas las ciudades de la lista. En el archivo `bestPathCheck.c` van a encontrar código de soporte para probar el recorrido leyendo el archivo de entrada, incluso para generar un nuevo archivo con la heurística aplicada. Tengan en cuenta que para este ejercicio pueden modificar cualquier parte del código, incluso de la estructura de datos que pide el trabajo práctico. Es más, pueden usar incluso otros lenguajes a fin de generar una solución incluso más eficiente que les permita encontrar el mejor camino posible en un tiempo razonable. El resultado de este ejercicio es, el archivo `cities.csv` reordenado según el mejor recorrido propuesto y una explicación de como obtuvieron el camino.

Entregable

Para este trabajo práctico no deberán entregar un informe. Sin embargo, deben agregar comentarios en el código que expliquen su solución. No deben comentar qué es lo que hace cada una de las instrucciones sino cuáles son las ideas principales del código implementado y por qué resuelve cada uno de los problemas.

La entrega debe contar con el mismo contenido que fue dado para realizarlo más lo que ustedes hayan agregado, habiendo modificado **solamente** los archivos `utils.c` y `main.c`. Es requisito para aprobar entregar el código **correctamente comentado**.

Lista de archivos provistos:

- `Makefile`: Script para compilar el trabajo práctico. Se ejecuta por medio del comando '`make`'
- `main.c`: Archivo donde completar la solución de los casos de test pedidos.
- `utils.c`: Archivo donde completar todo el código a implementar.
- `utils.h`: Encabezados, definiciones de estructuras y funciones.
- `bestPathCheck.c`: Programa que carga las ciudades de ejemplo y verifica la longitud total. también permite aplicar una heurística para mejorar el orden entre las ciudades.
- `ciudades.csv`: Lista de ciudades de ejemplo para utilizar en `bestPathCheck.c`.