# Algoritmos y Estructura de Datos

## Tercer Parcial

*Planificación de vuelo de un drone*

Facultad de Ciencias Exactas, Físicas y Naturales
Universidad Nacional de Córdoba, Argentina
Junio, 2021


**Docentes:**
Wolfmann, Gustavo
Ayme, Rubén

**Alumnos:**
Lorenzo, Facundo; DNI: 41411627
Riba, Franco; DNI: 42638706
Roig, Patricio; DNI: 41411655

**Grupo:** 32

En este informe se describe la lógica y estrategias utilizadas para la implementación realizada tomando como base los contenidos vistos en clases teóricas y prácticas. Se incluye el detalle de las estructuras de datos utilizadas y los distintos módulos cuyos comportamientos conjuntos logran cumplir de forma satisfactoria con la consigna planteada.

A partir de un archivo .txt completamos una **matriz de arreglos de bits**, la cual indica donde existen manchas o barreras. A partir de esta matriz se verifica que manchas poseen caminos directos mutuamente y generamos una **matriz de adyacencia** y otra de **pesos**. Posteriormente utilizamos la matriz de adyacencia para encontrar todos los **ciclos hamiltonianos** posibles y con la matriz de pesos hallamos el de menor coste, es decir el más óptimo.

## Resumen de funciones en función main

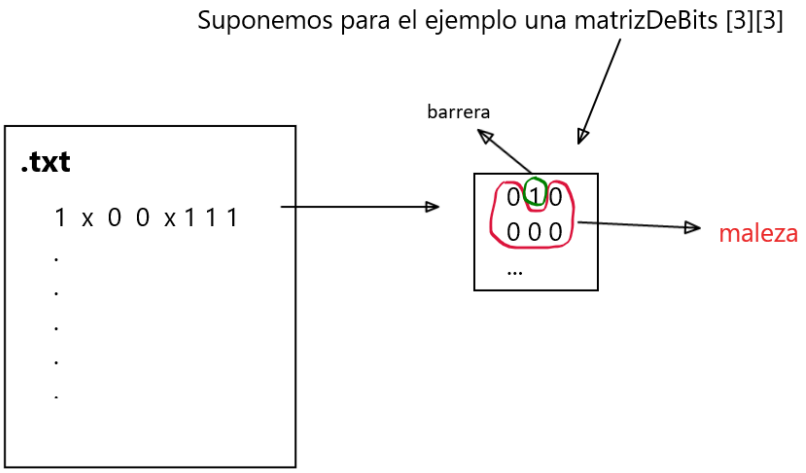
| Función | Tipo | Descripción |
|---|---|---|
| llenarMatrizDeBits() | void | Se completa una matriz de bits (arreglo de bit sets) en base a los caracteres en un archivo de texto (que representa la imagen de 100x100 px). El valor de los bits es:<br>0 si hay maleza (un 1 en el .txt)<br>1 si hay barrera (una x en el .txt) |
| filtrar() | void | Función auxiliar. Recorre la matriz de bits, revisando cada elemento y decide:<br>0 -> LLamado a función revisarManchas(int y, int x)<br><br>1 -> LLamado a función revisarBarreras(int y, int x) |
| revisarManchas(int y, int x) | void | Encuentra las coordenadas (x,y) del centro de manchas de al menos 3x3 |
| revisarBarreras(int y, int x) | void | Genera variable tipo barrera (modelada como recta vertical u horizontal) con información sobre las coordenadas (xi,yi) y (xf,yf) de inicio y final de la barrera. |
| completarMatrizAdyacencia() | void | Genera la matriz de adyacencia que representa las aristas de un grafo, es decir las conexiones existentes o inexistentes entre distintos nodos/ vértices |

| | | |
|---|---|---|
| doIntersect(Posicion p1, Posicion q1, Posicion p2, Posicion q2) | bool | Retorna true hay una intersección entre las dos barreras analizadas y retorna false si no existe dicha intersección.<br>Adaptado desde: How to check if two given line segments intersect? - GeeksforGeeks |
| orientation(Posicion p, Posicion q, Posicion r) | int | "Decodifica" la orientación de una barrera en base a sus puntos de destino y final<br>Adaptado desde: How to check if two given line segments intersect? - GeeksforGeeks |
| onSegment(Posicion p, Posicion q, Posicion r) | bool | Dados 3 puntos colineales, la función verifica si el punto "q" está sobre un segmento "pr"<br>Adaptado desde: How to check if two given line segments intersect? - GeeksforGeeks |

## Descripción detallada de funciones más importantes

### void llenarMatrizDeBits()

Se completa la matriz *bitset<2> matrizDeBits[FILAS][COLUMNAS]* leyendo los caracteres del archivo de texto y saltando a rellenar una nueva fila cada vez que se alcanza la cantidad de columnas configuradas y comprobando si se corresponden con un "1" o con una "x", en el primer caso el valor seteado en la matriz de bits en la posición (i,j) será un 0(hay maleza), mientras que en el segundo se setearia un 1 (hay barrera).




### void revisar()

Esta función simplemente recorre la matriz de bits, revisando cada elemento y siguiendo el siguiente criterio:

- Si el elemento es un 0 -> LLamado a función revisarManchas(int y, int x)

- Si el elemento es un 1 -> LLamado a función revisarBarreras(int y, int x)

**void revisarManchas(int y, int x)**

Esta función comprueba que exista un área de al menos 3x3 alrededor de la posición (representada por los índices “filas”, “columnas” de la matriz de bits) de la mancha. En caso de que esto ocurra se crea una variable “posicion” del tipo Posicion (Struct) que contiene la coordenada del centro (sobre la parte más arriba a la izquierda en caso de una área mayor a 3x3) de cada región de manchas de al menos 3x3.

Una vez “marcada” una mancha se resetean los datos en ciertas posiciones para que no se produzcan conflictos en verificaciones posteriores.

**void revisarBarreras(int y, int x)**

En esta función se crea una variable tipo “Barrera” (Struct) accesible mediante un puntero “barrera” , internamente esta estructura tienen dos variables tipo “Posición” que indican el punto de origen y el punto final de una recta.

Inicialmente se fija el punto de origen (xref) con las coordenadas recibidas como parámetro en base a las filas y columnas de la matriz de bits.

Primero de verifica la existencia de una barrera de forma horizontal:

Se corrobora en base a la matriz de bits si la **posición a la derecha** de las coordenadas parámetro corresponde también a una barrera.

- Si corresponde con una barrera se incrementa un contador que **indica la cantidad de posiciones contiguas** a la posición de referencia que también son barreras. Se utiliza una bandera booleana para seguir realizando este proceso iterativamente hasta hallar el final de la barrera.
- Si no se corresponde con una barrera se setea el punto final de la recta teniendo en cuenta el contador mencionado en el punto anterior:

  xfinal = xref + contador

  yfinal = yref

A continuación se resetea la matriz del mapa de bits en aquellas posiciones (en este caso hacia la derecha de la referencia) ocupadas por la barrera para no volver a utilizarlas.

En segunda instancia se verifica la existencia de una barrera vertical de forma análoga al proceso anterior.

Se corrobora en base a la matriz de bits si la **posición superior** a la de las coordenadas parámetro corresponde también a una barrera.

- Si corresponde con una barrera se incrementa un contador que **indica la cantidad de posiciones hacia arriba** de la posición de referencia que también son barreras. Se utiliza una bandera booleana para seguir realizando este proceso iterativamente hasta hallar el final de la barrera.
- Si no se corresponde con una barrera se setea el punto final de la recta teniendo en cuenta el contador mencionado en el punto anterior:

  xfinal = xref

  yfinal = yref + contador

A continuación se resetea la matriz del mapa de bits en aquellas posiciones (en este caso hacia arriba) ocupadas por la barrera para no volver a utilizarlas.

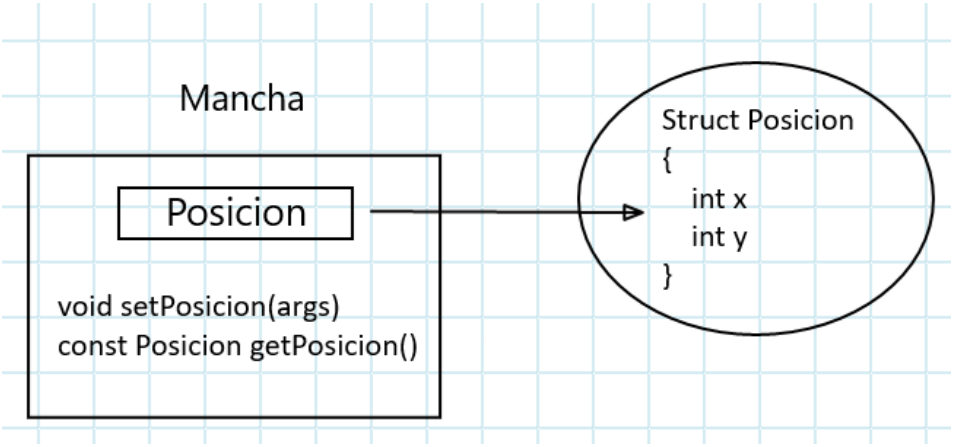
Una vez finalizado este proceso se dispone de una variable tipo barrera (modelada como recta vertical u horizontal) con información sobre las coordenadas (xi,yi) y (xf,yf) de inicio y final de la barrera.

**void completarMatrizAdyacencia();**

Se inicializa la matriz de adyacencia con “todas barreras” mediante un bucle iterativo.

Se completan las posiciones con 1 si los vértices adyacentes están interconectados (si hay camino), para esto utilizamos los métodos revisarBarrera y revisarMancha.

**Clase Mancha**



## Estructura Barrera

Esta estructura modela una recta representativa de las barreras existentes en el terreno, por lo tanto se define de la siguiente manera:

```
struct Barrera{
    Posicion p1;
    Posicion p2;
};
```

Donde las variables de tipo “Posición” representan las coordenadas de los puntos de inicio y final de la barrera (recta) respectivamente.,

## Clase Hamilton

En esta clase se incorporan todos los métodos principales y auxiliares para hallar todos los caminos hamiltonianos, y posteriormente, analizarlos desde el programa principal para hallar el  más corto de ellos, es decir el óptimo.

### Resumen de métodos en  clase Hamilton

MAX NODOS: variable global equivalente al número de nodos existentes.

| Función | Tipo | Descripción |
|---|---|---|
| isSafe(int v, int graph[MAXNODOS][MAXNODOS], vector<int> path , int pos) | bool | Esta función se usa para verificar si un vértice puede ser añadido en la posición “pos” en el ciclo hamiltoniano. |
| hamCycle(int graph[MAXNODOS][MAXNODOS]) | void | En esta función hallamos todos los ciclos hamiltonianos posibles. |
| FindHamCycle() | void | Esta función es utilizada de forma auxiliar por hamCycle para hallar algún ciclo hamiltoniano. |

Una vez que se tienen todos los ciclos hamiltonianos guardados en una matriz, comenzamos a buscar el camino más corto, para ello comparamos la distancia que recorre el camino actual de la iteración, con el anterior camino recorrido, en caso de que el camino actual sea menor que el anterior se actualiza el menor.

# Conclusión

La realización de este proyecto nos ayudó a comprender con mucha más profundidad el contenido teórico y práctico visto en las clases, de forma que pudimos integrar lo aprendido para lograr nuestra implementación, con respecto a esta última, en general estamos satisfechos, pero creemos que con algo más de tiempo se podría haber logrado un resultado más completo y genérico. Por último, estamos muy satisfechos con los conocimientos adquiridos durante todo el cursado ya que hemos notado la gran importancia que toman estos contenidos.