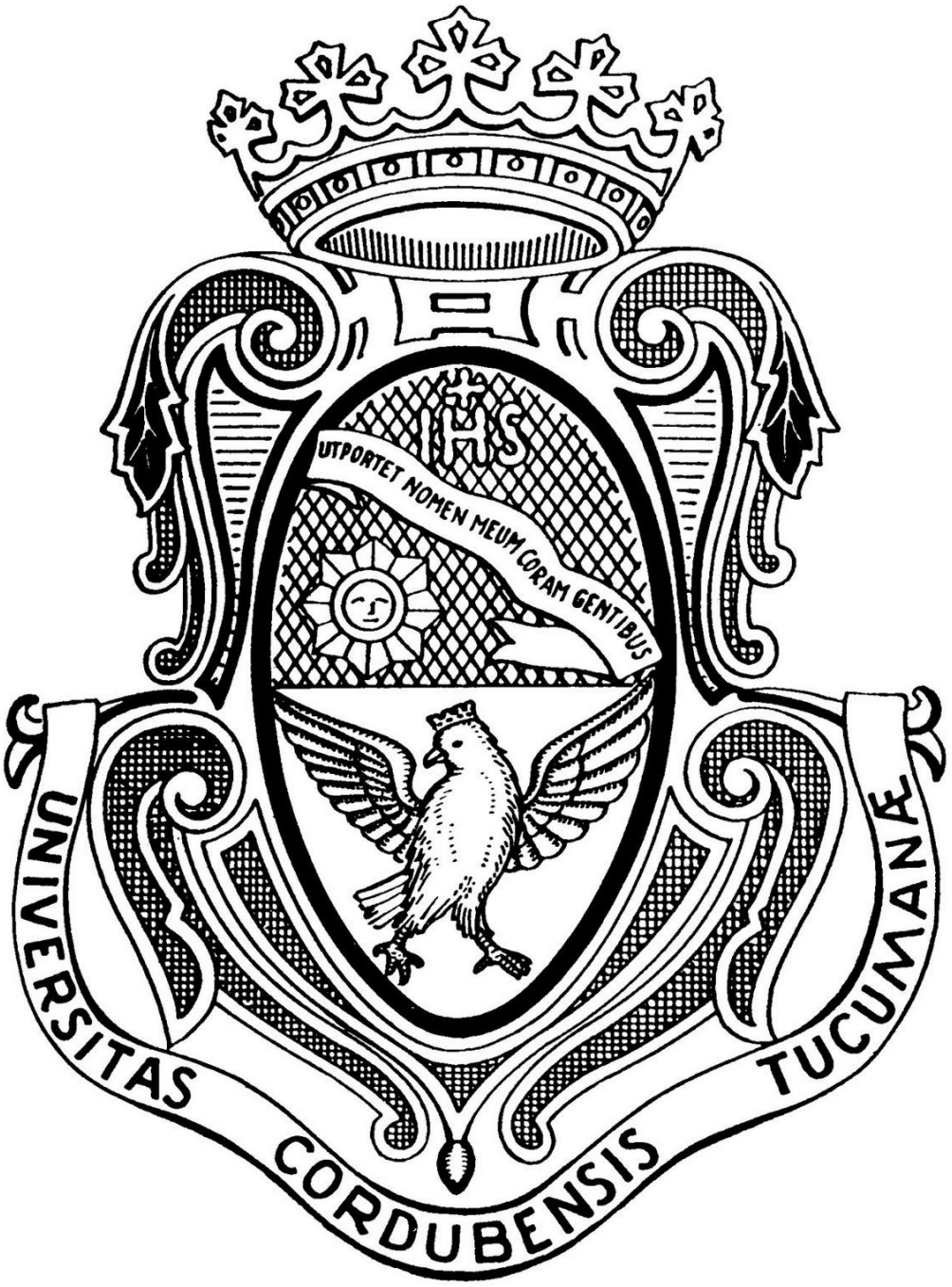
# Algoritmos y Estructuras de Datos
# Parcial 3




Alumnos:

| | |
|---|---|
| Natale, Alfredo | 95558906 |
| Pasolli, Nestor Jeremias | 42853888 |
| Melia, Nicolas Osvaldo | 43167517 |

**Introducción** **3**
Enunciado 3

**Desarrollo** **5**
Graph 5
Header Tree 7

**Conclusión** **9**

**Apéndice: Solución alternativa** **10**
Header Variables 10
Header Metodos 11

# Introducción

## Enunciado

Los alumnos que promocionan Algoritmos y Estructuras de Datos están planeando realizar un viaje a Europa para festejar la promoción. El plan es llegar a Madrid y hacer un recorrido por 8 ciudades en total, regresando a Madrid. Como el presupuesto es limitado, no deben repetir ciudades y además analizar varias opciones para tomar una decisión de como hacer el recorrido más económico. El costo de los tramos entre ciudades es simétrico.

Las ciudades a visitar son (junto a sus iniciales):

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Madrid | MA | París | PA | Roma | RO | Zurich | ZU |
| Berlín | BE | Amsterdam | AM | Varsovia | VA | Viena | Vi |

Los costos de los tramos es el siguiente:

| | MA | RO | PA | AM | ZU | BE | VA | VI |
|---|---|---|---|---|---|---|---|---|
| MA | INF | INF | 210 | INF | 300 | INF | 350 | 320 |
| RO | INF | INF | 350 | INF | 150 | INF | 280 | 250 |
| PA | 210 | 350 | INF | 100 | INF | 120 | INF | INF |
| AM | INF | INF | 100 | INF | 200 | 60 | INF | INF |
| ZU | 300 | 150 | INF | 200 | INF | 120 | INF | 240 |
| BE | INF | INF | 120 | 60 | 120 | INF | 60 | 80 |
| VA | 350 | 280 | INF | INF | INF | 60 | INF | 60 |
| VI | 320 | 250 | INF | INF | 240 | 80 | 60 | INF |

La representación de los costos es triangular ya que el grafo es no dirigido. Filas es origen y columnas destino. El problema se denomina tradicionalmente como “ciclo Hamiltoneano”. Puede ver un video explicatorio del algoritmo en https://www.youtube.com/watch?v=-AX-U6Ok0is

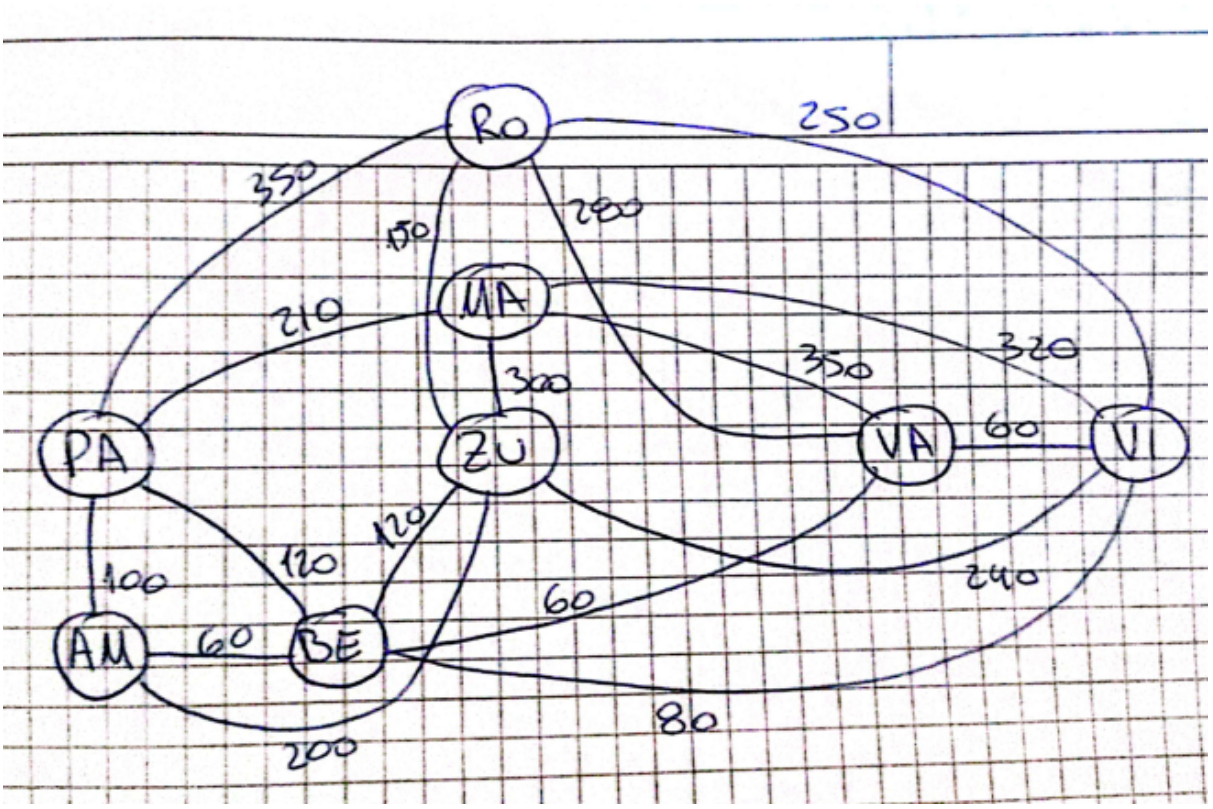
Sin embargo, dadas las múltiples conexiones entre las ciudades deseadas, puede haber más de un ciclo Hamiltoneano posible, por lo que deberá encontrar todos los ciclos posibles y determinar el de menor costo.

Para encontrar todos los ciclos, deberá aplicar un algoritmo de búsqueda en amplitud, utilizando un árbol m-ario que tenga como raíz un punto de partida y cada hijo represente la próxima ciudad que puede accederse y que sea una opción válida a los fines del ciclo.

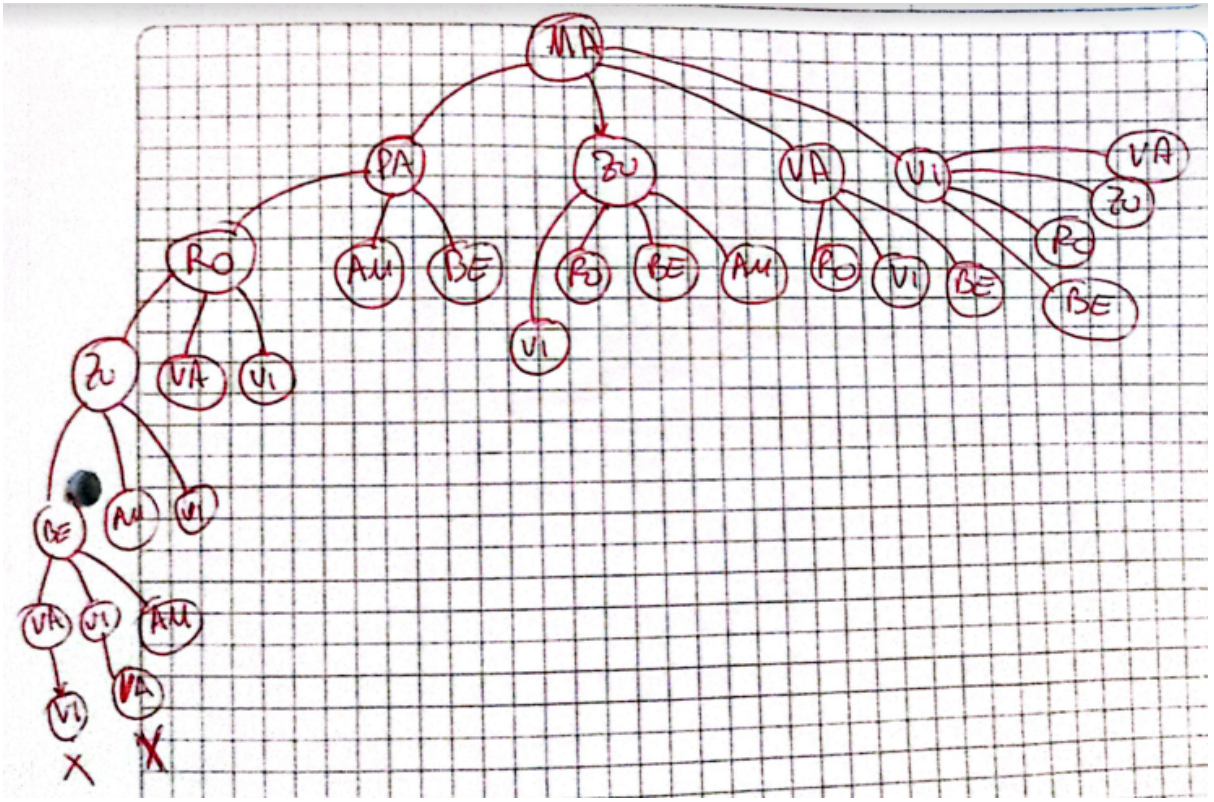
Tenga en cuenta las propiedades que deben cumplir los ciclos Hamiltoneanos expuestos en el link anterior ya que permite minimizar la cantidad de opciones a evaluar y por ende, el tiempo de procesamiento total del algoritmo.

Para todos los ciclos Hamiltoneanos determinados debe imprimir la secuencia de ciudades que lo conforman y el costo total del mismo.

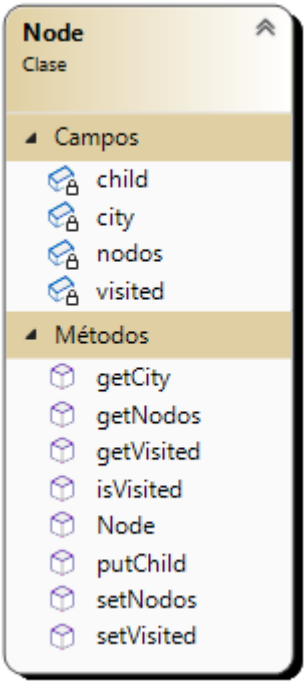
En un primer momento, para analizar el problema y comenzar a plantear una solución, dibujamos el grafo en cuestión representado por la matriz de adyacencia de la consigna. El grafo es el siguiente:




Además, para visualizar cómo el algoritmo debe ir buscando caminos y construyendo así un árbol m-ario que represente en sus ramas las distintas posibilidades de recorridos, realizamos un dibujo que muestra simplemente una parte de este árbol, a modo de ayuda visual, como se ve a continuación:


MA
PA
ZU
VA
VI
VA
ZU
PO
BE
RO
AM
BE
PO
BE
AM
PO
VI
BE
BE
VI
ZU
VA
VI
BE
AM
VI
VA
VI
AM
VA
VI
X
X

# Desarrollo




El proyecto consta de dos archivos: el archivo “graph.cpp” y un header “Tree.hpp”

## Graph

En este archivo encontramos el método main, como así también distintas funciones que se encargarán de recorrer el grafo por amplitud.

```
int main() {
    int grafo[CANT][CANT] = { INF, INF, 210, INF, 300, INF, 350, 320,
                              INF, INF, 350, INF, 150, INF, 280, 250,
                              210, 350, INF, 100, INF, 120, INF, INF,
                              INF, INF, 100, INF, 200, 60, INF, INF,
                              300, 150, INF, 200, INF, 120, INF, 240,
                              INF, INF, 120, 60, 120, INF, 60, 80,
                              350, 280, INF, INF, INF, 60, INF, 60,
                              320, 250, INF, INF, 240, 80, 60, INF };



    bfs(first:0, graph:grafo);
    cout << "\n\n\nEl recorrido optimo es ";
    for (int i = 0; i < CANT; i++) {
        cout << indexCities[optimo.first[i]] << " - ";
    }
    cout << "Madrid\nCon un costo de " << optimo.second << endl;
```

```
void bfs(int first, int graph[CANT][CANT]) {
    int contador = 0;
    queue<Node*> toVisit;
    bool visitado[CANT] = {1, 0, 0, 0, 0, 0, 0, 0};
    vector<int> nodos;
    nodos.push_back(_Val:first);
    Node* node = new Node(first, visitado, nodos);
    toVisit.push(_Val:node);

    while (!toVisit.empty()) {
        node = toVisit.front(); toVisit.pop();
        for (int i = 0; i < CANT; i++) { ... }
    }
}
```

```
bool isHamiltoniano(Node prueba, int graph[8][8]) {
    queue<Node*> toVisit;
    Node* node;
    vector<int> nodos;
    toVisit.push(_Val:&prueba);

    while (!toVisit.empty()) {
        node = toVisit.front(); toVisit.pop();
        for (int i = 0; i < CANT; i++) { ... }
    }
    return false;
}
```

```
void presupuesto(vector<int> recorrido, int graph[8][8]) {
    int presupuesto = 0;

    for (int i = 0; i < 7; i++) {
        presupuesto += graph[recorrido[i]][recorrido[i + 1]];
    }
    presupuesto += graph[recorrido[7]][0];

    if (optimo.second == 0 || presupuesto < optimo.second) {
        optimo.first = recorrido;
        optimo.second = presupuesto;
    }

    cout << "El presupuesto de ir por ese recorrido es: " << presupuesto << endl << endl;
```

- *main()*: En primer lugar inicializa la matriz de adyacencia que simboliza nuestro grafo en cuestión. Luego llama al método *bfs()*, que es el que contiene la lógica para buscar los ciclos hamiltonianos. Una vez estos ciclos fueron encontrados, se calcula cual es el óptimo, teniendo en cuenta los costos de las aristas, y lo almacena en un *pair<vector<int>, int>*, que contiene el vector que indica el orden en el que se recorren las ciudades y el costo, respectivamente.

- *bfs(int , int[][])*: Este método es el que realiza la búsqueda en amplitud sobre un árbol m-ario de los caminos. Esto lo hace conceptualmente, ya que no implementa un árbol en la memoria como tal, sino que va creando diferentes nodos y buscando

posibles “hijos” utilizando un método llamado “isHamiltoniano”, que corresponden a nodos que cumplen las condiciones necesarias para formar un ciclo hamiltoniano. Una vez que valida que un posible “hijo” cumple las condiciones, lo agrega a este “árbol”.

- *isHamiltoneano(Node,int[][])*: se encarga de buscar, mediante BFS, si el nodo pasado como argumento posee un ciclo Hamiltoniano asociado. Una vez encontrado este, retorna true, caso contrario retorna false. El valor de retorno es utilizado por la función *bfs* para decidir si es viable o no colocar dicho nodo como posible recorrido en el “árbol”.

- presupuesto(vector<int>, int [][]): recibe como parámetros un vector que contiene un recorrido válido, y el grafo. Se encarga de calcular el costo total de ese camino en cuestión, y compararlo con el costo menor guardado hasta el momento, en caso de encontrar que este nuevo costo es menor, lo reemplaza y lo guarda como el camino de costo mínimo encontrado hasta el momento.

## Header Tree

Este header contiene la clase “Node”, que posee los atributos necesarios para cada nodo de nuestro árbol, y los métodos necesarios para manejar estos atributos.

```
class Node {
private:
    int city;
    bool visited[CANT];
    vector<Node*> child;
    vector<int> nodos;
public:
    Node(int ciudad, bool visitado[CANT], vector<int> n) {
        city = ciudad;
        for (int i = 0; i < CANT; i++) visited[i] = visitado[i]; nodos = n;
    }
    int getCity() { return city; }
    void putChild(Node* n) { child.push_back(_Val:n); }
    bool isVisited(int n) { if (visited[n]) return true; return false; }
    bool* getVisited() { return visited; }
    void setVisited(int n) { visited[n] = true; }
    void setNodos(int i) { nodos.push_back(_Val:i); }
    vector<int> getNodos() { return nodos; }
};
```

- *int city*: número entero que representa a una de las ciudades.

- *bool visited[]*: arreglo que tiene valores “true” en las ciudades ya visitadas y “false” en las que no fueron visitadas.

- *vector<Node*> child*: vector que guarda los hijos del nodo en cuestión, esto es lo que simula la estructura de un árbol, ya que cada nodo tendrá un vector con sus hijos.

- *vector<int> nodos*: vector que guarda el recorrido realizado hasta el nodo en cuestión.
- *Node(int,bool,vector<int>)*: El constructor de la clase se utiliza para crear una instancia de Node, recibiendo un entero que simboliza la ciudad a almacenar, un array booleano que simboliza los nodos ya visitados hasta el momento, y un *vector<int>* que almacena el recorrido Hamiltoniano.
- *putChild(Node*)*: agrega un hijo al nodo en cuestión, es decir lo agrega al vector "child".
- *int getCity()*: retorna el valor entero correspondiente a la ciudad almacenada en ese nodo.
- *isVisited(int)*: recibe un valor entero correspondiente a una ciudad y retorna true o false. Dichos valores de retorno se utilizan para decidir si la ciudad pasada como parámetro fue o no visitada.
- *bool* getVisited()*: retorna el arreglo visited[].
- *setVisited(int)*: setea algún elemento de arreglo visited[], es decir, marca una ciudad como ya visitada.
- *setNodos(int)*: almacena en el vector nodos el nodo pasado como parámetro a la función
- *vector<int> getNodos()*: retorna el vector "nodos", que contiene el recorrido realizado hasta el momento.

# Conclusión

- Durante la realización de este trabajo práctico pudimos comprender cómo podemos utilizar los grafos para representar situaciones y problemas de la vida real, y con este plantear posibles soluciones.
- Pudimos comprender el funcionamiento de los distintos algoritmos para recorrer grafos. Tales son BFS(Breadth-First-Search) y DFS(Depth-First-Search).
- Pudimos comparar la complejidad de implementar distintas soluciones al mismo problema utilizando cada uno de estos algoritmos.

# Apéndice: Solución alternativa

Probando posibles soluciones al problema encontramos una solución que utiliza una recursión para analizar posibles ciclos, es decir haciendo una búsqueda en profundidad por el grafo. Esta solución fue descartada por no cumplir con la consigna, que requería una búsqueda en amplitud, pero nos pareció interesante debido a su simplicidad en el código. Cabe aclarar que es menos eficiente desde el punto de vista computacional, ya que al ser en profundidad analiza todos los posibles caminos uno por uno, validando cada caso al final, mientras que la búsqueda en amplitud va descartando posibilidades inválidas en cada iteración.

Para la implementación de esta solución alternativa utilizamos tres ficheros: Variables.h, Metodos.h, main.cpp

## Header Variables

Este fichero contiene las variables usadas durante la ejecución del programa, y el grafo.

```
#define INF 9000
#define CANT 8
```

```
int visitado[CANT] = { [0]: 0, [1]: 0, [2]: 0, [3]: 0, [4]: 0, [5]: 0, [6]: 0, [7]: 0 };
int recorrido[CANT+1] = { [0]: -1, [1]: -1, [2]: -1, [3]: -1, [4]: -1, [5]: -1, [6]: -1, [7]: -1, [8]: -1 };
pair<int[8], int>optimo;


int grafo[CANT][CANT] = { INF, INF, [0].[2]: 210, INF, [0].[4]: 300, INF, [0].[6]: 350, [0].[7]: 320,
                          INF, INF, [1].[2]: 350, INF, [1].[4]: 150, INF, [1].[6]: 280, [1].[7]: 250,
                          [2].[0]: 210, [2].[1]: 350, INF, [2].[3]: 100, INF, [2].[5]: 120, INF, INF,
                          INF, INF, [3].[2]: 100, INF, [3].[4]: 200, [3].[5]: 60, INF, INF,
                          [4].[0]: 300, [4].[1]: 150, INF, [4].[3]: 200, INF, [4].[5]: 120, INF, [4].[7]: 240,
                          INF, INF, [5].[2]: 120, [5].[3]: 60, [5].[4]: 120, INF, [5].[6]: 60, [5].[7]: 80,
                          [6].[0]: 350, [6].[1]: 280, INF, INF, INF, [6].[5]: 60, INF, [6].[7]: 60,
                          [7].[0]: 320, [7].[1]: 250, INF, INF, [7].[4]: 240, [7].[5]: 80, [7].[6]: 60, INF };

string indexCities[CANT] = { [0]: "Madrid", [1]: "Roma", [2]: "Paris", [3]: "Amsterdan", [4]: "Zurich",
                             [5]: "Berlin", [6]: "Varsovia", [7]: "Viena" };
```

Podemos encontrar las siguientes variables:

- visitado[CANT]: este es un array de enteros que guarda información de qué ciudades fueron visitadas durante la búsqueda de los ciclos Hamiltonianos.
- recorrido[CANT+1]: este array de enteros contiene la información del recorrido de ciudades de cada ciclo Hamiltoniano.

- pair<int[8], int>óptimo: esta estructura de datos almacena en su primer atributo el recorrido óptimo, y en el segundo atributo el costo que involucra ese recorrido.
- int grafo[CANT][CANT]: este array bidireccional contiene el grafo.
- string indexCities[CANT] : este array simplemente relaciona los índices de fila/columna con su ciudad correspondiente, por lo que es un arreglo de strings.
- INF 9000: simboliza que no hay arista entre una ciudad y otra en la matriz de adyacencia.
- CANT 8: es la cantidad de ciudades.

## Header Metodos

```
void buscarCiclo(int, int);
void recorrer();
int esCiclo();
void verCiclos();
void esOptimo(int arr[]);
```

```
void buscarCiclo(int nodo, int cantvisitados) {
    if (cantvisitados == CANT) {
        if (esCiclo()) {
            cout << "Encontre un ciclo Hamiltoneano";
            verCiclos();
            recorrido[CANT] = recorrido[0];
            recorrer();
            esOptimo( arr: recorrido);
        }

    }
    for (int i = 0; i < CANT; i++) { //i representa destino
        if (grafo[nodo][i] != INF && !visitado[i]) {
            visitado[i] = 1;
            recorrido[cantvisitados] = i;
            buscarCiclo( nodo: i,  cantvisitados: cantvisitados + 1);
            visitado[i] = 0;
        }
    }
}
```

```
void verCiclos() {
    for (int i = 0; i < 8; i++) {
        if (grafo[recorrido[i]][recorrido[i + 1]] == INF) cout << "\n----------
    }
}
```

```
void esOptimo(int arr[]) {
    int presupuesto = 0;

    for (int i = 0; i < 8; i++) {
        presupuesto += grafo[recorrido[i]][recorrido[i + 1]];
    }

    if (optimo.second == 0) {
        for (int i = 0; i < 9; i++) {
            optimo.first[i] = arr[i];
        }
        optimo.second = presupuesto;
    }
    else if (presupuesto < optimo.second) {
        for (int i = 0; i < 9; i++) {
            optimo.first[i] = arr[i];
        }
        optimo.second = presupuesto;
    }

    cout << "el presupuesto de ir por ese recorrido es: " << presupuesto << endl << endl;

}
```

```
int esCiclo() {
    int inicial = recorrido[0];
    int final = recorrido[CANT - 1];
    return (grafo[final][inicial] != INF);
}
```

```
void recorrer() {
    int i;
    cout << "\n";
    for (i = 0; i < CANT; i++) {
        cout << indexCities[recorrido[i]] << " ";
    }
    cout << indexCities[recorrido[0]];
    cout << "\n";
}
```

Este fichero contiene los métodos utilizados durante la ejecución del programa. En este fichero encontraremos:

- buscarCiclo(int, int): este método es el encargado de buscar cada ciclo Hamiltoniano posible.
- recorrer(): este método se utiliza para recorrer el array ‘recorridos’ para luego imprimirlo en pantalla.
- esCiclo(): se encarga de verificar si el recorrido almacenado en ‘recorridos’ se trata de un ciclo Hamiltoniano.
- verCiclos(): se encarga de verificar si el posible ciclo Hamiltoniano encontrado es en sí mismo el esperado.
- esOptimo(int arr[]): este método recibe un array llamado ‘recorridos’ como argumento y se encarga de verificar si el nuevo recorrido encontrado es más óptimo que el ya encontrado previamente.