Memoria del proyecto final de Algoritmos y Estructuras de Datos

Lydia Ruiz Martínez y Jorge Vančo Sampedro

24-5-2023

Algoritmos y Estructuras de Datos

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

Índice

1. Descripción

En este documento se explican detalladamente el desarrollo y las estructuras de datos implementadas en el proyecto final de la asignatura de Algoritmos y Estructuras de Datos realizado por Lydia Ruiz Martínez y Jorge Vančo Sampedro, alumnos del Grado en Ingeniería Matemática e Inteligencia Artificial en la Escuela Técnica Superior de Ingeniería (ICAI) de la Universidad Pontificia Comillas.

El proyecto final consiste en un sistema de gestión de vuelos que permite encontrar todos los vuelos disponibles utilizando datos almacenados en un archivo CSV. Para lograr esto, se han implementado los siguientes elementos de desarrollo de software: 745 líneas de código, 10 módulos, 38 funciones, 9 clases, un archivo de datos y la librería pygame. Además, se han utilizado algunas estructuras de datos estudiadas en la asignatura, como un árbol n-ario para almacenar la información de los vuelos y una lista doblemente enlazada circular. También se ha empleado el algoritmo de Quicksort para ordenar los vuelos por precio de manera rápida y efectiva, de modo que el vuelo más barato sea el primero que se muestra al usuario.

En primer lugar, se realiza la lectura de un archivo CSV que contiene un total de 9.534.417 vuelos, cada uno con características asociadas como el origen, el destino, los kilómetros recorridos, el precio y el trimestre en el que se realiza el viaje. Después de leer los datos, se insertan en el árbol n-ario, el cual será explicado en detalle en la siguiente sección.

```python
def leer_datos(fichero: str, arbol: Arbol, caracteristicas: list[str], screen=None, N: int = None) -> dict:
    """
    Lee el fichero con los vuelos y los inserta en el arbol

    Args:
        fichero (str): el nombre del fichero

    Returns:
        dict_opciones_por_caracteristica (dict): diccionario que guarda todos los posibles valores que puede tomar cada característica
    """
    print("LEYENDO DATOS", end="")

    dict_opciones_por_caracteristica: dict = {}
    for caracteristica in caracteristicas:
        dict_opciones_por_caracteristica[caracteristica] = []
    j = 1
    with open(fichero, "r") as fh:

        for vuelo in data_reader(fh):

            arbol.insertar_vuelo(vuelo)

            for caracteristica in caracteristicas:
                valor = getattr(vuelo, caracteristica)
                if valor not in dict_opciones_por_caracteristica[caracteristica]:
                    dict_opciones_por_caracteristica[caracteristica].append(
                        valor)

            if screen != None and j % 500_000 == 0:
                mostrar_texto_medio(
                    screen, f"Cargando 9.534.417 vuelos... {j/9_534_417:.02%}")

            j += 1

        print("\nCarga de datos finalizada")

    return dict_opciones_por_caracteristica
```

Figura 1. Función que permite leer los datos del fichero y los añade al árbol.

2. Árbol

La estructura del árbol n-ario se ha implementado para guardar todos los vuelos que tienen las mismas características, siendo cada característica asociada al vuelo con los requisitos impuestos por el usuario un nodo del árbol.

```python
class Nodo():
    """
    Es la clase de cada Nodo que forma el Árbol
    """

    def __init__(self, orden: list[str], caracteristica: str = None, diccionario: dict = None, vuelos: list[Vuelo] = None) -> None:
        self.orden: list[str] = orden
        # ["origen", "destino", "compania", "trimestre"]
        self.caracteristica: str = caracteristica
        self.diccionario: dict[str, Nodo] = {
        } if diccionario == None else diccionario
        self.vuelos: list[Vuelo] = [] if vuelos == None else vuelos

    def __repr__(self) -> str:
        return f"Nodo(característica = {self.caracteristica})"
```

Figura 2. Clase Nodo del árbol.

```python
class Arbol():
    """
    Clase del árbol
    Este árbol es una estructura que sirve para guardar los vuelos que tienen las mismas características
    """

    def __init__(self, orden: list[str]) -> None:
        self.raiz: Nodo = None
        self.orden: list[str] = orden

    def insertar_vuelo(self, vuelo: Vuelo) -> None:
        """
        Añade el vuelo en su correcto lugar en el árbol

        Args:
            vuelo (Vuelo): El objeto del vuelo a añadir
        """
        if self.raiz == None:  # Se crea la raíz del nodo
            self.raiz: Nodo = Nodo(self.orden, self.orden[0])

        nodo: Nodo = self.raiz
        # mira si está la caracteristica del vuelo en el diccionario del vuelo  |  ver si vuelo.origen in nodo.dict
        posicion: int = 0
        for caracteristica in self.orden:  # Recorre el árbol a partir de las características
            atributo: str = getattr(
                vuelo, caracteristica)   # Atributo del vuelo
            if atributo not in nodo.diccionario:
                if posicion + 1 >= len(self.orden):
                    nuevo_nodo = Nodo(self.orden)
                else:
                    nuevo_nodo = Nodo(self.orden, self.orden[posicion+1])
                nodo.diccionario[atributo] = nuevo_nodo

            nodo = nodo.diccionario[atributo]

            posicion = posicion + 1

        nodo.vuelos.append(vuelo)
```

Figura 3. Clase Árbol.

```python
def mostrar(self, nodo: Nodo = None, rama: str = "") -> None:
    """
    Muestra la lista de vuelos de cada rama y el camino hasta ella

    Args:
        nodo (Nodo): El nodo desde el que se muestra (no poner nada para que empiece desde la raíz)
        rama (str): La string con el camino que lleva

    Returns:
        None
    """
    if nodo == None:  # Coge la raíz si no se le pasa ningún nodo
        nodo: Nodo = self.raiz

    if nodo:  # Para comprobar que el árbol no esté vacío
        for key, nodo_hijo in nodo.diccionario.items():
            if nodo_hijo.vuelos:  # Si el nodo hijo tiene los vuelos, muestra la rama
                print(rama + "->"+key+":", len(nodo_hijo.vuelos), "vuelos")
            else:  # Si el nodo hijo no tiene los vuelos, vuelve a mostrar desde él mismo
                self.mostrar(nodo_hijo, rama + "->" + str(key))

def _get_vuelos_ramas(self, nodo: Nodo, vuelos: list[Vuelo] = None) -> list:
    """
    Busca los buelos de las ramas de un nodo

    Args:
        nodo (Nodo): El nodo desde que se quiere buscar todos los vuelos
        vuelos (list): La lista con los vuelos que ya se han añadido

    Returns:
        vuelos (list)
    """
    if vuelos == None:  # Se crea la lista de vuelos
        vuelos = []

    for nodo_hijo in nodo.diccionario.values():  # Se recorren los nodos hijos
        if nodo_hijo.vuelos:
            vuelos += nodo_hijo.vuelos
        else:
            # Se buscan los vuelos desde hijos del nodo
            self._get_vuelos_ramas(nodo_hijo, vuelos)

    return vuelos
```

Figura 4. Métodos de la clase Arbol.

```python
def buscar(self, filtros: list[str]) -> list[Vuelo]:
    """
    Busca los vuelos según ciertos filtros

    Args:
        filtos (list): lista con los valores de las caracteristicas de los vuelos que se quieren
    Returns:
        vuelos (list): lista de los vuelos
    """
    nodo = self.raiz
    i: int = 0
    while i < len(filtros) and i >= 0:  # Recorre los nodos a partir de los filtros
        filtro = filtros[i]
        caracteristica = nodo.caracteristica
        if filtro:
            nodo = nodo.diccionario.get(filtro, None)
        if nodo:
            i += 1
        else:
            i = -1

    if i == -1:
        mensaje = f"No hay ningún vuelo con esas características"
        return None, mensaje

    if nodo.vuelos:  # Si tiene los vuelos, se devuelven
        vuelos = nodo.vuelos
    else:
        # Si no tiene los vuelos, devuelve todos los vuelos de las ramas de ese nodo
        vuelos = self._get_vuelos_ramas(nodo)
    return vuelos, ""
```

Figura 5. Métodos de la clase Árbol.

3. Lista doblemente enlazada circular

En el proyecto también se utiliza una lista doblemente enlazada circular como estructura de datos adicional. Esta lista se ha creado con el propósito de mostrar al usuario los vuelos disponibles una vez que ha completado los campos obligatorios.

La implementación de la lista doblemente enlazada circular resulta muy beneficiosa para la visualización de los vuelos. Mediante el uso de punteros, cada vuelo está conectado tanto al vuelo siguiente como al vuelo anterior, ya que se trata de una lista enlazada bidireccional. Esta característica facilita la navegación entre los vuelos, permitiendo al usuario avanzar o retroceder de manera sencilla. Además, gracias a que la lista es circular, una vez que el usuario llega al último vuelo, puede acceder rápidamente al primer vuelo de la lista, cerrando así el ciclo de manera eficiente.

```python
class NodoLista:
    def __init__(self, vuelo: Vuelo, id: int) -> None:
        self.vuelo: Vuelo = vuelo
        self.next: NodoLista = None
        self.prev: NodoLista = None
        self.id: int = id

    def get_id(self):
        return self.id

    def __repr__(self) -> str:
        return f"vuelo: {self.vuelo}"
```

Figura 6. Clase NodoLista usada en la clase ListaDoble.

```python
class ListaDoble:
    """
    Lista doblemente enlazada y circular
    """

    def __init__(self) -> None:
        self.head: NodoLista = None
        self.tail: NodoLista = None
        self.n_nodos = 0

    def insertar_vuelo(self, vuelo: Vuelo) -> None:
        """
        Inserta el vuelo al final de la lista

        Args:
            vuelo (Vuelo): el vuelo a insertar
        """
        self.n_nodos += 1
        nodo_vuelo = NodoLista(vuelo, self.n_nodos)
        if self.head == None:
            self.head = nodo_vuelo
            self.tail = nodo_vuelo
        else:
            nodo_vuelo.next = self.head
            nodo_vuelo.prev = self.tail
            self.tail.next = nodo_vuelo
            self.head.prev = nodo_vuelo
            self.tail = nodo_vuelo

    def RecorrerListaCabeza(self) -> None:
        print("RECORRIDO DE LISTA DESDE LA CABEZA: ")
        if self.head is None:
            print("No existen elementos para recorrer")
        else:
            aux = self.head
            while aux:
                print(aux)
                if self.tail == aux:
                    aux = False
                else:
                    aux = aux.next
```

Figura 7. Clase ListaDoble.

4. Quicksort

Con el fin de ordenar los vuelos de menor a mayor precio, se ha utilizado el algoritmo de ordenación Quicksort. Elegimos este algoritmo por su gran eficiencia en términos de tiempos de ejecución y para reducir la necesidad de memoria adicional, pues ordena los elementos en su lugar y no requiere espacio adicional para almacenar los vuelos ordenados.

```python
def divide(vuelos: list[Vuelo], iz: int, de: int) -> int:
    """
    Encuentra la posición de partición en el array
    Args:
        vuelos: El array a ordenar
        iz: El índice izquierdo
        de: El índice derecho
    """
    pivote = vuelos[de].precio  # Elige como pivote el elemento A[de]

    i = iz - 1  # Inicializa el puntero

    for j in range(iz, de):
        if vuelos[j].precio <= pivote:

            i = i + 1  # Mueve el puntero
            # Intercambia el elemento i con el j
            (vuelos[i], vuelos[j]) = (vuelos[j], vuelos[i])

    # Intercambia el pivote con el elemento de la posición i+1
    (vuelos[i + 1], vuelos[de]) = (vuelos[de], vuelos[i + 1])

    return i + 1
```

Figura 8. Función auxiliar para dividir la lista con los vuelos.

```python
def quickSort(vuelos: list[Vuelo], iz: int = 0, de: int = None) -> list[Vuelo]:
    """
    Función principal que implementa el algoritmo de Quicksort
    Args:
        vuelos: El array a ordenar
        iz: El índice izquierdo
        de: El índice derecho
    """
    if not vuelos:  # Por si no hay vuelos
        return

    if de == None:
        de = len(vuelos)-1

    if iz < de:

        piv = divide(vuelos, iz, de)  # Para encontrar la posición de partición

        quickSort(vuelos, iz, piv - 1)  # Para ordenar el lado izquierdo

        quickSort(vuelos, piv + 1, de)  # Para ordenar el lado derecho
    return vuelos
```

Figura 7. Función del algoritmo de Quicksort.

5. Ejemplos de ejecución

En esta sección, se presenta de manera detallada la ejecución del sistema con su respectiva interfaz gráfica. La experiencia comienza con la etapa de carga de datos, donde se procesa y se incorpora al sistema toda la información relativa a los vuelos disponibles.

Figura 8. Carga de vuelos.

Una vez finalizada la carga de los vuelos, se brinda al usuario la posibilidad de especificar las características deseadas para su vuelo. En este punto, se le solicita que ingrese datos como el origen, destino, trimestre en el que se realiza el viaje y compañía. Es importante mencionar que todos los campos anteriormente mencionados a excepción de la compañía son obligatorios para garantizar la precisión de los resultados.

Si el usuario no completa todos los campos obligatorios, el sistema mostrará una notificación o mensaje de aviso, asegurando que se proporcionen los datos requeridos antes de proceder con la búsqueda. Esta medida contribuye a evitar resultados incompletos.

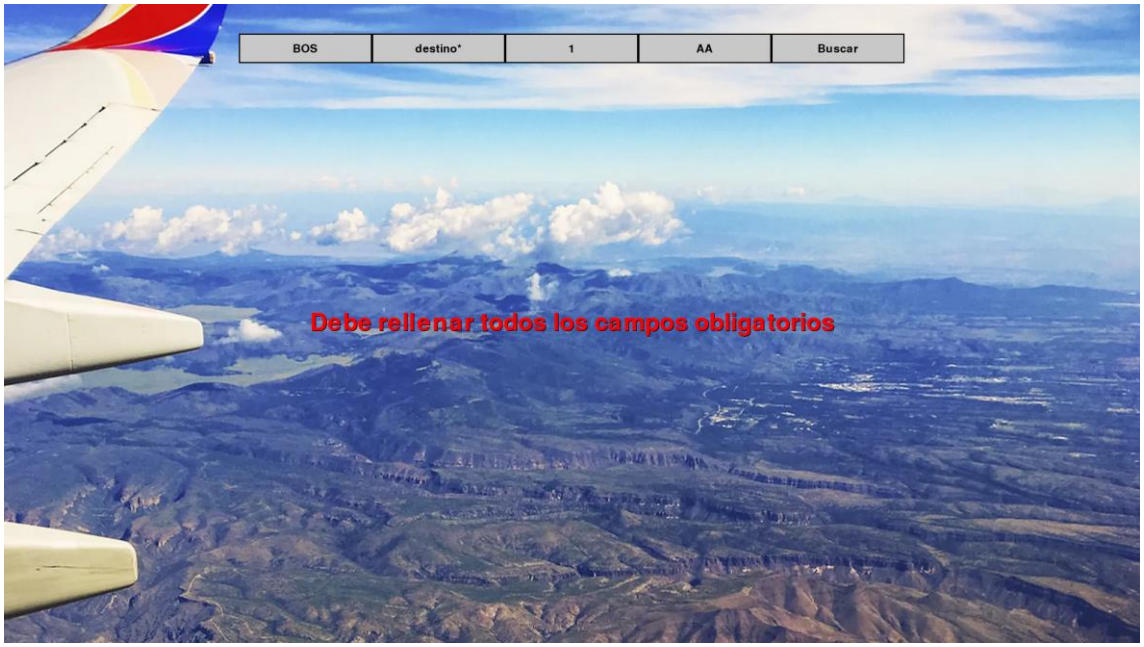

Figura 9. Destino no seleccionado.

Una vez que se han ingresado todas las características deseadas y se ha validado la integridad de los datos, el sistema realizará una búsqueda exhaustiva en la base de datos de vuelos. Utilizando algoritmos eficientes y estructuras de datos optimizadas, se filtran y seleccionan aquellos vuelos que cumplen con las condiciones requeridas por el usuario.

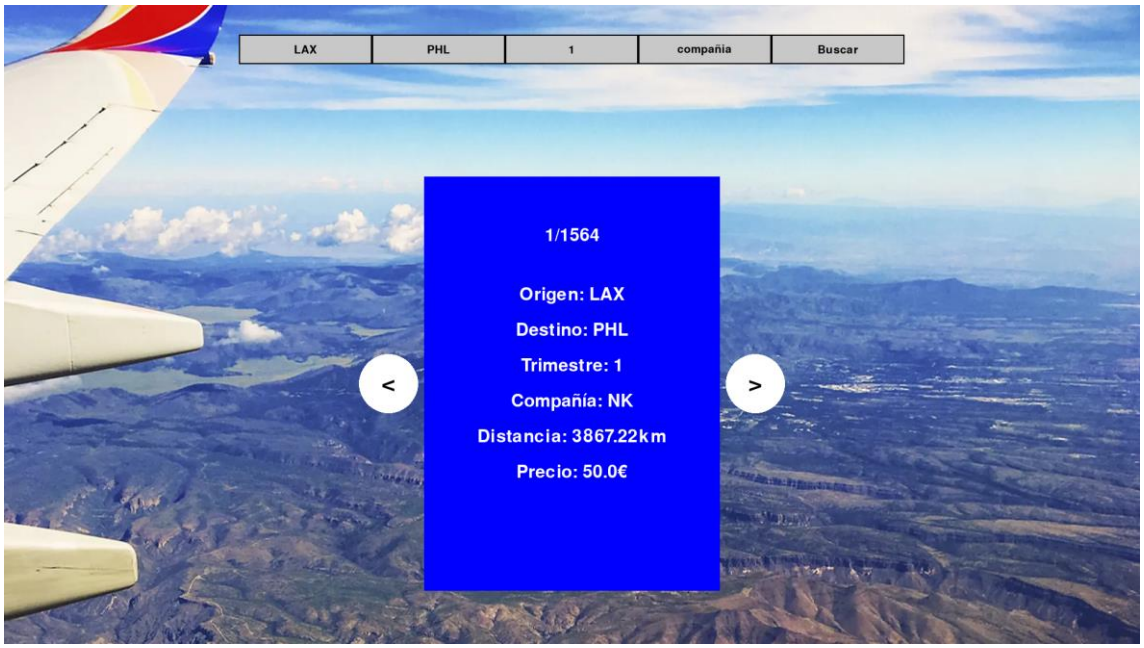

Figura 10. Primer vuelo (de precio menor) con las características seleccionadas.