**Ramificación y Poda**

Nombre: Jorge

Apellidos: Martinez Pazos

DNI: 39510046W

Correo: jmartinez5741@alumno.uned.es

Centro Asociado: Vigo (Pontevedra)

Preguntas Teóricas

## Indica y razona sobre el coste temporal y espacial del algoritmo desarrollado.

COSTE TEMPORAL:

El algoritmo de Ramificación y Poda se basa en explorar el espacio de soluciones mediante una estrategia de búsqueda en la que se generan nodos (subproblemas) y se podan ramas no prometedoras para mejorar la eficiencia. En cada paso, se evalúa si una rama del espacio de búsqueda debe ser descartada en función de ciertos criterios de optimalidad o viabilidad, lo cual permite reducir el número de nodos explorados.

* En el **peor caso**, el algoritmo podría explorar todas las posibles soluciones, lo que implicaría un coste exponencial en función del número de nodos a explorar, es decir, O(n^k), donde **n** es el factor de ramificación (número de hijos por nodo) y **k** es la profundidad del árbol de búsqueda. Esto sucede cuando no hay poda o cuando la poda no es suficientemente eficiente.
* Sin embargo, con una poda eficaz, el algoritmo reduce considerablemente el número de nodos que explora, haciendo que el coste temporal real dependa de la eficiencia de la poda. En la práctica, la complejidad temporal podría reducirse considerablemente dependiendo de la naturaleza del problema y las restricciones de poda.

COSTE ESPACIAL**:**

El coste espacial en un algoritmo de Ramificación y Poda está determinado principalmente por el almacenamiento de los nodos activos que están siendo explorados en el momento dado, ya sea en una lista, cola o montículo.

* En el peor caso, si no se realiza poda o esta es ineficaz, el coste espacial podría ser O(n^k), ya que cada nodo en el espacio de búsqueda debe mantenerse en memoria mientras se procesa.
* Sin embargo, si la poda es efectiva, el número de nodos en memoria se reduce significativamente. Además, el espacio necesario para almacenar información adicional (como las soluciones parciales o las restricciones de poda) también puede afectar al coste espacial, pero en general este es menor que el coste temporal.

En resumen, el coste temporal es **exponencial** en el peor caso (O(n^k)), pero puede mejorar con una poda efectiva, mientras que el coste espacial depende de la cantidad de nodos activos almacenados y puede reducirse si la poda es eficiente.

## Explica razonadamente las diferencias existentes frente a una implementación utilizando Vuelta Atrás y la selección de la mejor solución encontrada.

Vuelta Atrás (Backtracking):

La **Vuelta Atrás** es una técnica de búsqueda exhaustiva en la que se exploran todas las posibles soluciones de un problema, descartando aquellas que no cumplen con las restricciones a medida que se avanza en la construcción de la solución. A medida que se avanza por el espacio de búsqueda, si en algún momento se llega a una configuración que no es válida o que no puede llevar a una solución óptima, se retrocede para explorar otras posibilidades.

Diferencias clave entre Ramificación y Poda y Vuelta Atrás:

1. **Estrategia de exploración:**
   * **Ramificación y Poda**: Divide el espacio de soluciones en subproblemas y utiliza una función de poda para eliminar ramas enteras que no pueden llevar a una mejor solución que la mejor encontrada hasta el momento. La poda mejora significativamente la eficiencia del algoritmo, ya que evita la exploración de soluciones no viables.
   * **Vuelta Atrás**: Explora todas las posibles configuraciones de forma exhaustiva, retrocediendo cuando se encuentra una solución parcial que no es válida. Aunque utiliza poda en forma de retroceso, no aprovecha una poda global como en la Ramificación y Poda, lo que lleva a una exploración más amplia del espacio de soluciones.
2. **Tamaño del espacio de búsqueda:**
   * **Ramificación y Poda**: El espacio de búsqueda se reduce considerablemente gracias a la poda. Aunque en el peor caso puede ser exponencial (O(n^k)), la poda permite eliminar muchas ramas, lo que hace que el algoritmo sea más eficiente.
   * **Vuelta Atrás**: El espacio de búsqueda es mucho más grande, ya que el algoritmo tiene que explorar todas las configuraciones posibles, lo que podría resultar en una complejidad O(n^k), sin la capacidad de eliminar soluciones de forma tan efectiva como en Ramificación y Poda.
3. **Coste temporal:**
   * **Ramificación y Poda**: El coste temporal depende de la poda. Si la poda es eficiente, el número de nodos a explorar disminuye considerablemente, lo que mejora el rendimiento. En el peor caso, el coste temporal sigue siendo O(n^k), pero en la práctica, con una poda efectiva, el tiempo de ejecución es mucho menor que el de una implementación de Vuelta Atrás.
   * **Vuelta Atrás**: Aunque puede ser más fácil de implementar, el algoritmo de Vuelta Atrás tiene un mayor coste temporal en comparación con Ramificación y Poda, ya que explora más configuraciones antes de encontrar una solución válida. El coste temporal suele ser O(n^k) sin optimización.
4. **Coste espacial:**
   * **Ramificación y Poda**: El coste espacial depende del número de nodos activos que se mantienen en memoria. Con una poda eficaz, el número de nodos activos puede reducirse significativamente, haciendo que el coste espacial sea menor en comparación con Vuelta Atrás.
   * **Vuelta Atrás**: En términos de espacio, la Vuelta Atrás generalmente requiere más memoria, ya que necesita mantener en memoria todas las soluciones parciales y retroceder a ellas, lo que genera un mayor coste espacial.

Resumen de las diferencias:

* **Ramificación y Poda**: Utiliza poda para reducir el espacio de búsqueda y mejorar la eficiencia tanto en términos de tiempo como de espacio.
* **Vuelta Atrás**: Es más exhaustiva y menos eficiente, ya que explora todas las posibles configuraciones sin la poda global que optimiza Ramificación y Poda.

## Explica y justifica –si procede– si el uso de una lista ordenada en lugar de un montículo para almacenar los nodos puede mejorar la eficiencia del algoritmo.

Lista ordenada vs. Montículo:

* **Montículo (Heap)**: Un montículo es una estructura de datos especializada que permite insertar nodos en O(log n) y extraer el nodo con el valor mínimo (o máximo) también en O(log n). El montículo se utiliza típicamente en algoritmos de Ramificación y Poda donde es necesario acceder de manera eficiente al nodo más prometedor (por ejemplo, el nodo con el coste más bajo) en cada iteración.
* **Lista ordenada**: Una lista ordenada mantiene los elementos en orden, lo que permite una extracción eficiente del nodo más pequeño (O(1) si se utiliza un acceso secuencial). Sin embargo, insertar un nodo en la lista ordenada requiere mantener el orden, lo que implica un coste de O(n) en el peor caso.

Comparación de eficiencia:

* **Inserción**:
  + Un **montículo** permite insertar nodos en tiempo O(log n), lo que es mucho más eficiente que insertar en una lista ordenada, que requiere O(n) en el peor caso.
* **Extracción**:
  + En una **lista ordenada**, la extracción del nodo más pequeño se realiza en O(1), lo cual es eficiente si el número de extracciones es bajo.
  + En un **montículo**, la extracción también se realiza en O(log n), que es más costoso que la extracción en una lista ordenada, pero el coste de inserción es significativamente menor.

**¿Es más eficiente usar una lista ordenada?** En general, **un montículo es más eficiente** que una lista ordenada para la mayoría de los algoritmos de Ramificación y Poda. Aunque la lista ordenada tiene una extracción más rápida (O(1)), la inserción en la lista ordenada es mucho más costosa (O(n)). Dado que en estos algoritmos se insertan y extraen nodos de manera frecuente, el montículo proporciona una mayor eficiencia global debido a sus tiempos de inserción más rápidos.

**Conclusión:** El uso de un **montículo** mejora la eficiencia del algoritmo en términos de coste temporal y espacial, especialmente cuando se insertan y extraen nodos con frecuencia. Una lista ordenada podría ser útil en algunos casos donde las extracciones son mucho más frecuentes que las inserciones, pero en general, un montículo es la opción más eficiente para la Ramificación y Poda.

Ejemplos de Ejecución

Posibles casos:

1. -t entrada.txt salida.txt
2. -h entrada1.txt salida1.txt (dan igual la entrada y salida)
3. entrada1.txt (inexistente) salida1.txt
4. -t entrada2.txt
5. -t
6. -x (argumento no valido)

Nos ubicamos con estos archivos en un inicio:Captura de pantalla de computadora

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Procedemos a pasar los argumentos de ejecución y lo probamos:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Obtenemos toda la traza (en la imagen solo se muestra una parte) y la salida adecuada:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Vamos a la segunda prueba:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Se obtiene correctamente la ayuda al pasar el argumento -h.

Procedemos a la siguiente prueba, con una entrada inexistente y una salida:

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Ahora vamos a probar sin aportar una salida, para que la muestre por consola:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Teniendo entrada2 los mismos valores que entrada para facilitar la corrección:

Imagen que contiene Texto

Descripción generada automáticamente

Y obtenemos:

Texto

Descripción generada automáticamente con confianza media

La traza, indicada por el argumento -t y la salida por consola al final.

En esta prueba procedo a pasarla únicamente -t como argumento, para que me pida los datos por consola y me muestre el resultado por ahí también:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

En la última prueba le pasaremos un argumento inválido para que salte error:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Código Fuente

## Clase MAIN:

package com.uned.pec1.asignaciontareas;

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.util.Scanner;

/\*\*

\* @author jorge

\*/

public class Main {

public static void main(String[] args) {

boolean traza = false;

String ficheroEntrada = null;

String ficheroSalida = null;

try {

// Procesar argumentos

for (String arg : args) {

if (arg.equals("-t")) {

traza = true;

} else if (arg.equals("-h")) {

mostrarAyuda();

return;

} else if (arg.endsWith(".txt")) {

if (ficheroEntrada == null) {

ficheroEntrada = arg;

} else if (ficheroSalida == null) {

ficheroSalida = arg;

} else {

throw new IllegalArgumentException("Demasiados archivos proporcionados.");

}

} else {

throw new IllegalArgumentException("Argumento no válido: " + arg);

}

}

// Leer parámetros

int[][] matrizCostes;

if (ficheroEntrada != null) {

matrizCostes = leerParametros(ficheroEntrada);

} else {

matrizCostes = leerParametrosDesdeConsola();

}

// Ejecutar el algoritmo

String resultado = AsignadorTareas.ejecutarAlgoritmo(matrizCostes, traza);

// Guardar o mostrar el resultado

if (ficheroSalida != null) {

escribirResultado(ficheroSalida, resultado);

} else {

System.out.println(resultado);

}

} catch (IllegalArgumentException e) {

System.err.println("Error: " + e.getMessage());

mostrarAyuda();

} catch (Exception e) {

System.err.println("Error inesperado: " + e.getMessage());

}

}

// Leer los parametros con archivo

private static int[][] leerParametros(String ficheroEntrada) throws IOException {

try (BufferedReader br = new BufferedReader(new FileReader(ficheroEntrada))) {

// Leer las dimensiones de la matriz desde la primera línea

String[] dimensiones = br.readLine().split(" ");

int filas = Integer.parseInt(dimensiones[0]);

int columnas = Integer.parseInt(dimensiones[1]);

// Crear la matriz de costes

int[][] matriz = new int[filas][columnas];

// Leer las filas de la matriz

for (int i = 0; i < filas; i++) {

String[] valores = br.readLine().split(" ");

for (int j = 0; j < columnas; j++) {

matriz[i][j] = Integer.parseInt(valores[j]);

}

}

return matriz;

} catch (IOException e) {

System.out.println("No se encontro el archivo de entrada, por lo que se procedera a pedir los datos por consola.");

return leerParametrosDesdeConsola();

}

}

// Leer los parametros sin archivo

private static int[][] leerParametrosDesdeConsola() {

Scanner scanner = new Scanner(System.in);

System.out.println("Introduce el numero de agentes de la matriz:");

int filas = scanner.nextInt();

System.out.println("Introduce el numero de tareas de la matriz:");

int columnas = scanner.nextInt();

int[][] matriz = new int[filas][columnas];

System.out.println("Introduce los valores de la matriz (fila por fila):");

for (int i = 0; i < filas; i++) {

System.out.println("Introduce los valores de la fila " + (i + 1) + " separados por espacios:");

for (int j = 0; j < columnas; j++) {

matriz[i][j] = scanner.nextInt();

}

}

return matriz;

}

private static void escribirResultado(String ficheroSalida, String resultado) throws IOException {

try (BufferedWriter bw = new BufferedWriter(new FileWriter(ficheroSalida))) {

bw.write(resultado);

}

}

private static void mostrarAyuda() {

System.out.println("SINTAXIS: java -jar tareas.jar [-t][-h] [fichero entrada] [fichero salida]");

System.out.println("-t Traza del algoritmo.");

System.out.println("-h Muestra esta ayuda.");

System.out.println("[fichero entrada] Archivo con la tabla de costes.");

System.out.println("[fichero salida] Archivo para guardar los resultados (opcional).");

}

}

## Clase MATRIZ:

/\*

\* Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license

\* Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template

\*/

package com.uned.pec1.asignaciontareas;

import java.io.\*;

/\*\*

\* @author jorge

\*/

public class Matriz {

private final String rutaArchivo;

private int filas;

private int columnas;

private int[][] tabla;

/\*\*

\* Constructor principal de la clase Matriz.

\*/

public Matriz(String rutaArchivo) {

this.rutaArchivo = rutaArchivo;

File file = new File(rutaArchivo);

if (!file.exists()) {

throw new IllegalArgumentException("El archivo no existe en la ruta especificada");

}

}

/\*\*

\* Constructor principal de la clase Matriz a partir de otra Matriz.

\*/

public Matriz(int[][] matrizCostes) {

this.rutaArchivo = null;

this.filas = matrizCostes.length;

this.columnas = matrizCostes[0].length;

this.tabla = matrizCostes;

}

// Devuelve la tabla

public int[][] obtenerTabla() {

return tabla;

}

// Leer la Matriz desde el archivo

public void cargarDesdeArchivo() throws IOException {

try (BufferedReader lector = new BufferedReader(new FileReader(rutaArchivo))) {

String[] dimensiones = lector.readLine().split(" ");

if (dimensiones.length != 2) {

throw new IllegalArgumentException("El archivo no tiene las dimensiones correctamente especificadas.");

}

filas = Integer.parseInt(dimensiones[0]);

columnas = Integer.parseInt(dimensiones[1]);

tabla = new int[filas][columnas];

for (int i = 0; i < filas; i++) {

String[] valores = lector.readLine().split(" ");

if (valores.length != columnas) {

throw new IllegalArgumentException("El numero de columnas no coincide con las dimensiones especificadas.");

}

for (int j = 0; j < columnas; j++) {

tabla[i][j] = Integer.parseInt(valores[j]);

}

}

}

}

// Guardar en archivo salida el resultado

public void guardarEnArchivo(String datos, String rutaSalida) throws IOException {

try (BufferedWriter escritor = new BufferedWriter(new FileWriter(rutaSalida))) {

escritor.write(datos);

}

}

// Pasar a texto el resultado obtenido

public String generarResultado(int[] asignaciones) {

StringBuilder resultado = new StringBuilder();

resultado.append("Asignaciones optimas:\n");

// Cambiar de Agente -> Tarea a Tarea -> Agente

for (int i = 0; i < asignaciones.length; i++) {

resultado.append("Tarea ").append(i + 1)

.append(" -> Agente ").append(asignaciones[i] + 1)

.append("\n");

}

return resultado.toString();

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

for (int[] fila : tabla) {

for (int valor : fila) {

sb.append(valor).append(" ");

}

sb.append("\n");

}

return sb.toString();

}

}

## Clase MONTICULO:

package com.uned.pec1.asignaciontareas;

/\*\*

\* @author jorge

\*/

public class Monticulo<E extends Comparable<E>> {

// Array para almacenar los elementos

private E[] elementos;

// Número actual de elementos

private int cantidad;

// Capacidad maxima

private int capacidad;

/\*\*

\* Constructor principal de la clase Monticulo.

\*/

public Monticulo(int capacidadMaxima) {

this.capacidad = capacidadMaxima;

this.cantidad = 0;

this.elementos = (E[]) new Comparable[capacidadMaxima];

}

public boolean estaVacio() {

return cantidad == 0;

}

// Agregar un elemento al monticulo

public void agregar(E elemento) {

if (cantidad == capacidad) {

throw new IllegalStateException("El monticulo esta lleno.");

}

elementos[cantidad] = elemento;

reorganizarHaciaArriba(cantidad);

cantidad++;

}

// Extraer el elemento de la cima

public E extraer() {

if (estaVacio()) {

throw new IllegalStateException("El monticulo esta vacio.");

}

E cima = elementos[0];

elementos[0] = elementos[--cantidad];

reorganizarHaciaAbajo(0);

return cima;

}

// Obtener el primer elemento sin extraerlo

public E primero() {

if (estaVacio()) {

return null;

}

return elementos[0];

}

// Reorganizar el elemento hacia arriba (flotar)

private void reorganizarHaciaArriba(int indice) {

int padre = (indice - 1) / 2;

while (indice > 0 && elementos[indice].compareTo(elementos[padre]) < 0) {

intercambiar(indice, padre);

indice = padre;

padre = (indice - 1) / 2;

}

}

// Reorganizar el elemento hacia abajo (hundir)

private void reorganizarHaciaAbajo(int indice) {

int menor = indice;

int hijoIzq = 2 \* indice + 1;

int hijoDer = 2 \* indice + 2;

if (hijoIzq < cantidad && elementos[hijoIzq].compareTo(elementos[menor]) < 0) {

menor = hijoIzq;

}

if (hijoDer < cantidad && elementos[hijoDer].compareTo(elementos[menor]) < 0) {

menor = hijoDer;

}

if (menor != indice) {

intercambiar(indice, menor);

reorganizarHaciaAbajo(menor);

}

}

// Intercambiar dos elementos

private void intercambiar(int i, int j) {

E temp = elementos[i];

elementos[i] = elementos[j];

elementos[j] = temp;

}

// Vaciar el montículo y establecer una nueva capacidad

public void vaciar(int capacidadMaxima) {

this.cantidad = 0;

this.capacidad = capacidadMaxima;

this.elementos = (E[]) new Comparable[capacidadMaxima];

}

// Verificar si el monticulo esta vacio

public boolean monticuloVacio() {

return cantidad == 0;

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder("Montículo: [");

for (int i = 0; i < cantidad; i++) {

sb.append(elementos[i]);

if (i < cantidad - 1) {

sb.append(", ");

}

}

sb.append("]");

return sb.toString();

}

}

## Clase NODO:

package com.uned.pec1.asignaciontareas;

import java.util.Arrays;

/\*\*

\* @author jorge

\*/

public class Nodo implements Comparable<Nodo> {

// Tareas asignadas

private final boolean[] tareasAsignadas;

// Agentes asignados a las tareas

private final int[] agentes;

// Coste acumulado

private int coste;

// Estimacion del coste optimo

private int estimacionOptima;

// Estimación del coste pesimista

private int estimacionPesimista;

// Valor usado para decidir si se poda

private int cota;

// Nivel del nodo en el arbol

private final int nivel;

/\*\*

\* Constructor principal de la clase Nodo.

\*/

public Nodo(boolean[] tareasAsignadas, int[] agentes, int coste, int estimacionOptima, int estimacionPesimista, int cota, int nivel) {

this.tareasAsignadas = Arrays.copyOf(tareasAsignadas, tareasAsignadas.length);

this.agentes = Arrays.copyOf(agentes, agentes.length);

this.coste = coste;

this.estimacionOptima = estimacionOptima;

this.estimacionPesimista = estimacionPesimista;

this.cota = cota;

this.nivel = nivel;

}

/\*\*

\* Constructor con copia. Crea un nuevo nodo hijo a partir de un nodo padre.

\*/

public Nodo(Nodo nodoPadre) {

this.tareasAsignadas = Arrays.copyOf(nodoPadre.tareasAsignadas, nodoPadre.tareasAsignadas.length);

this.agentes = Arrays.copyOf(nodoPadre.agentes, nodoPadre.agentes.length);

this.coste = nodoPadre.coste;

this.estimacionOptima = nodoPadre.estimacionOptima;

this.estimacionPesimista = nodoPadre.estimacionPesimista;

this.cota = nodoPadre.cota;

this.nivel = nodoPadre.nivel + 1;

}

// Crear un nodo hijo al asignar una nueva tarea a un agente.

public Nodo crearHijo(int agente, int[][] tablaCostes) {

boolean[] nuevasTareas = Arrays.copyOf(tareasAsignadas, tareasAsignadas.length);

int[] nuevosAgentes = Arrays.copyOf(agentes, agentes.length);

nuevasTareas[nivel] = true; // Marca la tarea actual como asignada

nuevosAgentes[nivel] = agente;

int nuevoCoste = coste + tablaCostes[agente][nivel];

int nuevaEstimacionOptima = calcularEstimacionOptima(tablaCostes, nuevasTareas, nivel + 1);

int nuevaEstimacionPesimista = calcularEstimacionPesimista(tablaCostes, nuevasTareas, nivel + 1);

int nuevaCota = Math.max(nuevaEstimacionOptima, nuevaEstimacionPesimista);

return new Nodo(nuevasTareas, nuevosAgentes, nuevoCoste, nuevaEstimacionOptima, nuevaEstimacionPesimista, nuevaCota, nivel + 1);

}

// Calcular la estimacion optima.

private int calcularEstimacionOptima(int[][] tablaCostes, boolean[] tareas, int nivelActual) {

int sumaMinimos = 0;

for (int i = nivelActual; i < tablaCostes.length; i++) {

int minimo = Integer.MAX\_VALUE;

for (int j = 0; j < tablaCostes[i].length; j++) {

if (!tareas[j]) {

minimo = Math.min(minimo, tablaCostes[i][j]);

}

}

sumaMinimos += minimo;

}

return sumaMinimos;

}

// Calcular la estimacion pesimista.

private int calcularEstimacionPesimista(int[][] tablaCostes, boolean[] tareas, int nivelActual) {

int suma = 0;

for (int tarea = nivelActual; tarea < tablaCostes.length; tarea++) {

int maximo = Integer.MIN\_VALUE;

for (int[] tablaCoste : tablaCostes) {

if (!tareas[tarea]) {

maximo = Math.max(maximo, tablaCoste[tarea]);

}

}

suma += maximo;

}

return suma;

}

// Comprobar si el agente está asignado

public boolean agenteAsignado(int agente) {

for (int asignado : agentes) {

if (asignado == agente) {

return true;

}

}

return false;

}

/\*\*

\* Métodos getter para acceder a los atributos del nodo.

\* @return

\*/

public int[] obtenerAsignaciones() {

return agentes;

}

public boolean tareaAsignada(int tarea) {

return tareasAsignadas[tarea];

}

public int getNivel() {

return nivel;

}

public int getCoste() {

return coste;

}

public int[] getAgentes() {

return agentes; // Un arreglo de enteros que representa la asignación de tareas

}

public boolean[] getTareasAsignadas() {

return tareasAsignadas;

}

public int getEstimacionOptima() {

return estimacionOptima;

}

public int getEstimacionPesimista() {

return estimacionPesimista;

}

public int getCota() {

return cota;

}

public void setCoste(int coste) {

this.coste = coste;

}

public void setEstimacionOptima(int estimacion) {

this.estimacionOptima = estimacion;

}

public void setEstimacionPesimista(int estimacion) {

this.estimacionPesimista = estimacion;

}

public void setCota(int cota) {

this.cota = cota;

}

@Override

public int compareTo(Nodo otro) {

return Integer.compare(this.estimacionOptima, otro.estimacionOptima);

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

sb.append("Asignaciones: ");

// Mostrar tareas primero y luego agentes

for (int i = 0; i < tareasAsignadas.length; i++) {

if (tareasAsignadas[i]) {

sb.append("Tarea ").append(i + 1).append(" -> Agente ").append(agentes[i] + 1).append("; ");

}

}

sb.append("\nCoste acumulado: ").append(coste);

sb.append(", Estimacion Optima: ").append(estimacionOptima);

sb.append(", Estimacion Pesimista: ").append(estimacionPesimista);

sb.append(", Cota: ").append(cota);

sb.append(", Nivel: ").append(nivel);

return sb.toString();

}

}

## Clase ASIGNADORTAREAS:

package com.uned.pec1.asignaciontareas;

import java.util.Arrays;

/\*\*

\* @author jorge

\*/

public class AsignadorTareas {

public static String ejecutarAlgoritmo(int[][] matrizCostes, boolean traza) {

try {

// Validar que la matriz no este vacia

if (matrizCostes.length == 0 || matrizCostes[0].length == 0) {

throw new IllegalArgumentException("La matriz de entrada está vacía.");

}

// Inicializar el algoritmo de Ramificacion y Poda

AlgoritmoRamificacionYPoda algoritmo = new AlgoritmoRamificacionYPoda(matrizCostes, traza);

// Inicializar las asignaciones y estimaciones

boolean[] tareasAsignadas = new boolean[matrizCostes.length];

int[] agentes = new int[matrizCostes.length];

Arrays.fill(agentes, -1);

// Calcular estimaciones iniciales

int estimacionOptima = 0;

int estimacionPesimista = 0;

for (int tarea = 0; tarea < matrizCostes[0].length; tarea++) {

int minimo = Integer.MAX\_VALUE;

int maximo = Integer.MIN\_VALUE;

for (int agente = 0; agente < matrizCostes.length; agente++) {

minimo = Math.min(minimo, matrizCostes[agente][tarea]);

maximo = Math.max(maximo, matrizCostes[agente][tarea]);

}

estimacionOptima += minimo;

estimacionPesimista += maximo;

}

// Crear el nodo raíz

Nodo nodoRaiz = new Nodo(

tareasAsignadas, // Ninguna tarea asignada

agentes, // Sin agentes asignados

0, // Coste inicial

estimacionOptima, // Estimación optimista inicial

estimacionPesimista, // Estimación pesimista inicial

0, // Nivel inicial

0 // Cota inicial

);

// Ejecutar el algoritmo

algoritmo.ejecutar(nodoRaiz);

// Validar si solucion

if (algoritmo.obtenerMejorSolucion() == null) {

return "No se encontro una solucion optima.";

}

// Generar el resultado en el formato esperado

// Matriz auxiliar para formatear el resultado

Matriz matriz = new Matriz(matrizCostes);

return matriz.generarResultado(algoritmo.obtenerMejorSolucion().obtenerAsignaciones());

} catch (IllegalArgumentException e) {

return "Error de validación: " + e.getMessage();

} catch (Exception e) {

e.printStackTrace();

return "Error inesperado durante la ejecucion del algoritmo.";

}

}

}

## Clase ALGORITMORAMIFICACIONYPODA:

package com.uned.pec1.asignaciontareas;

/\*\*

\* @author jorge

\*/

public class AlgoritmoRamificacionYPoda {

private final int[][] tablaCostes;

private final Monticulo<Nodo> colaPrioridad;

private Nodo mejorSolucion;

private final boolean mostrarTraza;

private final int[] minimos;

private final int[] maximos;

private int cota;

/\*\*

\* Constructor principal de la clase AlgoritmoRamificacionYPoda.

\*/

public AlgoritmoRamificacionYPoda(int[][] tablaCostes, boolean mostrarTraza) {

this.tablaCostes = tablaCostes;

this.colaPrioridad = new Monticulo<>(100);

this.mostrarTraza = mostrarTraza;

// Minimos de los costes

this.minimos = calcularExtremos(tablaCostes, true);

// Maximos de los costes

this.maximos = calcularExtremos(tablaCostes, false);

}

public void ejecutar(Nodo nodoInicial) {

// cota = EstimacionPes

cota = calcularEstimacion(maximos, nodoInicial, false);

// Insertar nodoRaiz en monticulo

colaPrioridad.agregar(nodoInicial);

// Con la traza activada

if (mostrarTraza) {

System.out.println("Ejecutando el algoritmo de Ramificacion y Poda.");

}

// Mientras nMonticuloVacio(monticulo)

while (!colaPrioridad.estaVacio()) {

Nodo nodoActual = colaPrioridad.extraer();

if (mostrarTraza) {

System.out.println("Procesando nodo en nivel " + nodoActual.getNivel() +

" con coste acumulado " + nodoActual.getCoste() +

" y estimacion optimista " + nodoActual.getEstimacionOptima());

}

// Si solucion

if (esSolucionCompleta(nodoActual)) {

// mejorSolucion <- hijo

actualizarMejorSolucion(nodoActual);

if (mostrarTraza) {

System.out.println("Mejor solucion provisional encontrada con coste " + mejorSolucion.getCoste());

}

} else {

// Insertar hijo en monticulo

expandirNodo(nodoActual);

}

}

}

// Nuevo nodo a insertar en el monticulo

private void expandirNodo(Nodo nodo) {

if (mostrarTraza) {

System.out.println("Expandiendo nodo en nivel " + nodo.getNivel() + " con coste " + nodo.getCoste());

}

int tarea = nodo.getNivel(); // La tarea corresponde al nivel actual

for (int agente = 0; agente < tablaCostes.length; agente++) {

if (!nodo.agenteAsignado(agente)) { // Si el agente no está asignado

Nodo nuevoNodo = new Nodo(nodo); // Crear una copia del nodo actual

nuevoNodo.getAgentes()[tarea] = agente; // Asignar el agente a la tarea actual

nuevoNodo.getTareasAsignadas()[tarea] = true; // Marcar la tarea como asignada

// Actualizamos el coste del hijo

nuevoNodo.setCoste(nodo.getCoste() + tablaCostes[agente][tarea]);

int estimacionOptima = calcularEstimacion(minimos, nuevoNodo, true);

nuevoNodo.setEstimacionOptima(Math.max(estimacionOptima, 0)); // Garantizamos que no sea negativa

int estimacionPesimista = calcularEstimacion(maximos, nuevoNodo, false);

nuevoNodo.setEstimacionPesimista(estimacionPesimista);

if (nuevoNodo.getEstimacionPesimista() < cota) {

nuevoNodo.setCota(nuevoNodo.getEstimacionPesimista());

}

if (nuevoNodo.getEstimacionPesimista() > cota) {

if (mostrarTraza) {

System.out.println("Nodo podado debido a su estimacion pesimista superior a la cota.");

}

continue; // Salta la expansión del nodo

}

if (nuevoNodo.getEstimacionOptima() > cota) {

if (mostrarTraza) {

System.out.println("Nodo podado debido a su estimacion optimista superior a la cota.");

}

continue; // Salta la expansión del nodo

}

colaPrioridad.agregar(nuevoNodo);

if (mostrarTraza) {

System.out.println("Nodo generado con asignacion de tarea " + (tarea + 1) +

" a Agente " + (agente + 1) + " con coste acumulado " + nuevoNodo.getCoste());

}

}

}

}

private boolean esSolucionCompleta(Nodo nodo) {

return nodo.getNivel() == tablaCostes[0].length;

}

private void actualizarMejorSolucion(Nodo nodo) {

if (mejorSolucion == null || nodo.getCoste() < mejorSolucion.getCoste()) {

mejorSolucion = nodo;

cota = mejorSolucion.getCoste(); // Actualizar la cota a la nueva mejor solución

}

}

private int calcularEstimacion(int[] valores, Nodo nodo, boolean optimista) {

int estimacion = nodo.getCoste();

for (int tarea = nodo.getNivel(); tarea < tablaCostes[0].length; tarea++) {

int valor = optimista ? minimos[tarea] : maximos[tarea];

estimacion += valor;

}

return estimacion;

}

public int[] calcularExtremos(int[][] tablaCostes, boolean buscarMinimos) {

int[] extremos = new int[tablaCostes[0].length];

for (int tarea = 0; tarea < tablaCostes[0].length; tarea++) {

int extremo = buscarMinimos ? Integer.MAX\_VALUE : Integer.MIN\_VALUE;

for (int agente = 0; agente < tablaCostes.length; agente++) {

if (buscarMinimos) {

extremo = Math.min(extremo, tablaCostes[agente][tarea]);

} else {

extremo = Math.max(extremo, tablaCostes[agente][tarea]);

}

}

extremos[tarea] = extremo;

}

return extremos;

}

public Nodo obtenerMejorSolucion() {

return mejorSolucion;

}

}