Actividad 3: Diseño de un Recorrido para el Cobrador

El problema descrito es una variante del Problema del Viajante (TSP), con la restricción adicional de que el cobrador no puede quedarse sin dinero en ningún momento. Para diseñar un recorrido que minimice el riesgo de quedarse sin dinero, podemos utilizar una combinación de heurísticas y restricciones. A continuación, te propongo un enfoque paso a paso:

1. Definición del Problema

Entrada:

Un conjunto de sucursales, cada una con un monto fijo que el cobrador puede cobrar (+) o pagar (-).

Un monto inicial de dinero con el que el cobrador comienza su recorrido.

Objetivo:

Planificar un recorrido que visite todas las sucursales sin quedarse sin dinero en ningún momento.

Priorizar sucursales donde se pueda cobrar más dinero para maximizar el saldo disponible.

2. Enfoque Propuesto

Para resolver este problema, podemos utilizar una heurística constructiva que priorice las sucursales con cobros más altos y evite las sucursales con pagos grandes cuando el saldo sea bajo. Aquí tienes los pasos detallados:

a) Ordenar las Sucursales

Clasificar las sucursales en dos grupos:

Sucursales con cobros (+): Aquellas donde el cobrador recibe dinero.

Sucursales con pagos (-): Aquellas donde el cobrador debe pagar.

Priorizar las sucursales con cobros más altos para maximizar el saldo disponible.

Evitar sucursales con pagos grandes cuando el saldo sea bajo.

b) Heurística de Selección

Regla de decisión:

En cada paso, el cobrador selecciona la sucursal no visitada que maximice su saldo disponible.

Si no hay sucursales con cobros disponibles, selecciona la sucursal con el pago más pequeño que pueda cubrir con su saldo actual.

c) Algoritmo Propuesto

Inicialización:

Comenzar con el monto inicial de dinero.

Marcar todas las sucursales como no visitadas.

Recorrido:

Mientras queden sucursales por visitar:

Calcular el saldo disponible.

Si hay sucursales con cobros (+):

Seleccionar la sucursal con el cobro más alto.

Si no hay sucursales con cobros (+):

Seleccionar la sucursal con el pago más pequeño que pueda cubrir con el saldo actual.

Actualizar el saldo y marcar la sucursal como visitada.

Finalización:

Si todas las sucursales han sido visitadas, el recorrido es válido.

Si en algún punto no se puede continuar, retroceder y probar otra sucursal (backtracking).

3. Ejemplo Práctico

Supongamos que el cobrador tiene un monto inicial de $100 y debe visitar las siguientes sucursales:

Sucursal Monto

A +50

B -30

C +20

D -40

E +10

Recorrido propuesto:

Inicio: Saldo = $100.

Visita A: Cobra

50.

S

a

l

d

o

=

50.Saldo=150.

Visita C: Cobra

20.

S

a

l

d

o

=

20.Saldo=170.

Visita E: Cobra

10.

S

a

l

d

o

=

10.Saldo=180.

Visita B: Paga

30.

S

a

l

d

o

=

30.Saldo=150.

Visita D: Paga

40.

S

a

l

d

o

=

40.Saldo=110.

El recorrido es válido porque el cobrador nunca se queda sin dinero.

4. Heurística Aplicada

La heurística utilizada es una combinación de:

Priorización de cobros: Maximizar el saldo disponible visitando primero las sucursales con cobros más altos.

Evitar pagos grandes: Seleccionar sucursales con pagos pequeños cuando el saldo es bajo.

5. Consideraciones Adicionales

Backtracking: Si en algún punto no se puede continuar, el algoritmo puede retroceder y probar otra sucursal.

Optimización: Para mejorar la eficiencia, se pueden utilizar técnicas como programación dinámica o metaheurísticas (por ejemplo, algoritmos genéticos).

Restricciones adicionales: Si hay restricciones de tiempo o distancia, se pueden incorporar al modelo.

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

// Principio de Responsabilidad Única (SRP)

class Sucursal {

private String nombre;

private double monto;

public Sucursal(String nombre, double monto) {

this.nombre = nombre;

this.monto = monto;

}

public String getNombre() {

return nombre;

}

public double getMonto() {

return monto;

}

}

// Principio de Abierto/Cerrado (OCP)

class PlanificadorRecorrido {

private double saldoInicial;

private List<Sucursal> sucursales;

public PlanificadorRecorrido(double saldoInicial, List<Sucursal> sucursales) {

this.saldoInicial = saldoInicial;

this.sucursales = new ArrayList<>(sucursales);

}

// Heurística: Ordenar sucursales por monto de cobro (de mayor a menor)

private void ordenarSucursales() {

Collections.sort(sucursales, Comparator.comparingDouble(Sucursal::getMonto).reversed());

}

// Planificar recorrido

public List<Sucursal> planificarRecorrido() {

ordenarSucursales();

List<Sucursal> recorrido = new ArrayList<>();

double saldoActual = saldoInicial;

for (Sucursal sucursal : sucursales) {

if (saldoActual + sucursal.getMonto() >= 0) {

recorrido.add(sucursal);

saldoActual += sucursal.getMonto();

}

}

return recorrido;

}

}

// Principio de Sustitución de Liskov (LSP) y Principio de Segregación de Interfaces (ISP)

interface Reporte {

void generarReporte(List<Sucursal> recorrido);

}

class ReporteTexto implements Reporte {

@Override

public void generarReporte(List<Sucursal> recorrido) {

System.out.println("Recorrido planificado:");

for (Sucursal sucursal : recorrido) {

System.out.println("Sucursal: " + sucursal.getNombre() + ", Monto: " + sucursal.getMonto());

}

}

}

// Principio de Inversión de Dependencias (DIP)

public class Main {

public static void main(String[] args) {

// Crear sucursales

List<Sucursal> sucursales = new ArrayList<>();

sucursales.add(new Sucursal("Sucursal A", 100));

sucursales.add(new Sucursal("Sucursal B", -50));

sucursales.add(new Sucursal("Sucursal C", 200));

sucursales.add(new Sucursal("Sucursal D", -30));

sucursales.add(new Sucursal("Sucursal E", 150));

// Planificar recorrido con saldo inicial de 100

PlanificadorRecorrido planificador = new PlanificadorRecorrido(100, sucursales);

List<Sucursal> recorrido = planificador.planificarRecorrido();

// Generar reporte

Reporte reporte = new ReporteTexto();

reporte.generarReporte(recorrido);

}

}