Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Práctica 4

Programación con ramificación y poda

Programación lineal

Jorge Solán Morote 816259

Francisco Javier Pizarro 821259

20/05/2023

Planteamiento del problema con ramificación y poda

Para este planteamiento el razonamiento a seguir es el siguiente: Partiendo de la situación inicial, es decir con todos los pedidos para elegir y con la capacidad disponible al máximo, debemos ir buscando en forma de árbol con un algoritmo Depth-First Search, es decir alcanzado al principio la primera solución completa para un recorrido en el árbol, es necesario emplear este algoritmo para poder realizar la poda de manera eficiente dado que una vez alcanzada una primera solución aunque no sea la más óptima nos ha dado un máximo inicial que va a poder descartar mediante la poda las peores soluciones.

En nuestro caso hemos decidido no implementar un TAD de árbol para ahorrar en tiempo de ejecución y en memoria, en su lugar hemos optado por el uso de una función de búsqueda recursiva DFS que actúa como “árbol”. Para evitar expandir nodos que no aportan una solución mejor simplemente cortamos la recursividad evitando así generar más nodos ineficientes.

Para afrontar las restricciones de capacidad del problema empleamos un array que registra el estado de la capacidad en cada etapa del problema, antes de generar una nueva llamada a un siguiente nodo comprobamos que este cumple las restricciones relativas a la capacidad con ayuda de este array.

Otro punto fundamental para la poda es el uso de una función de estimación tal que sin realizar demasiados cálculos devuelva una aproximación siempre positivista de la ganancia que es posible obtener en caso de seguir expandiendo nodos partiendo de la situación actual, en nuestro caso hemos decidido que dicha función para poder ser optimista calculará la ganancia total de añadir los pedidos disponibles de forma voraz(añadir primero los más valiosos) y teniendo en cuenta la capacidad total conjunta del sistema y no la capacidad individual de cada etapa del problema, de esta forma obtenemos un valor con una restricción más leve pero que es mayor que el valor real que podemos obtener.

Teniendo en cuenta las características de este primer planteamiento, hemos decidido emplear Golang como lenguaje de programación principal para resolverlo, dado que tiene un rendimiento muy cercano a C y nos ofrece ciertas funcionalidades de muy alto nivel tales como los Slices para trabajar cómodamente con vectores.

Planteamiento del problema con programación lineal

Para ser capaces de implementar una solución ya sea empleando la API o algún módulo que implemente algoritmos de programación lineal primero debemos plantear las ecuaciones que definen el problema, en este caso lo hemos planteado de la siguiente forma:

La función a maximizar en este problema es evidentemente el beneficio que se rige por la siguiente fórmula:

$$\sum_{i=0}^{p} pedido[i].pasajeros \ * \ (pedido[i].salida \ - \ pedido[i].llegada) \ * Y_i$$

En este caso el pedido contiene la información relativa a todos los pedidos es decir al realizar pedido[1] estamos accediendo a los datos del segundo pedido, cada pedido tiene en sus datos internos el número de pasajeros, la estación de salida y la estación de llegada. Para representar si un pedido ha sido elegido para ser recogido o no vamos a recurrir a las variables booleanas Y, cada pedido tiene su Y asociada.

Para plantear las restricciones del problema relativas a la capacidad vamos a emplear las variables X_J cada X_J , $J \in [1, m]$, el valor de X_J simboliza el estado del tren en la estación J, es decir el número de pasajeros que quedan dentro del tren al partir de dicha estación con esta definición la restricción es esta:

$$\forall X_J, J \in [1, m] | X_J <= n$$

En la anterior fórmula m simboliza el número de estaciones totales y n la capacidad máxima del tren.

Para definir que el valor de X_J ,es decir de la capacidad usada en cada estación, es igual al número de pasajeros que haya en ese momento implementamos la siguiente fórmula:

$$\forall X_J, J \in [1, m] | (\sum_{i=0}^{p} pedido[i].pasajeros * Y_i \ , pedido[i].salida <= J < pedido[i].llegada)$$

Es decir para cada X_J su valor debe ser igual al sumatorio del número de pasajero de cada uno de los pedidos que satisfagan las condiciones de haber sido elegidos y cumplir $J \in [pedido[i].salida, \ pedido[i].llegada)$.

Una vez hemos planteado el problema en forma de ecuaciones y restricciones debemos implementarlo mediante el uso de alguna API o módulo de programación lineal en nuestro caso hemos optado por recurrir al uso de un módulo disponible en Python concretamente el módulo Python MIP (Mixed-Integer Linear Programming) Tools.

El código implementado se encuentra en el anexo II.

Entorno de ejecución

Para poder ejecutar las soluciones el entorno de ejecución requiere:

- Python3
- Go

Antes de lanzar el programa mediante el script ejecutar.sh debemos tener instalado el módulo mip para python, este se instala mediante:

pip3 install mip

Tests

Las primeras pruebas realizadas han sido empleando para ambas soluciones los datos facilitados con el enunciado como ficheros de entrada. Para un análisis más profundo tanto a nivel de corrección como a nivel de rendimiento se han empleado ficheros custom para generar el contenido de dichos ficheros de forma pseudoaleatoria se ha empleado un pequeño script de ruby que se encuentra en el anexo III.

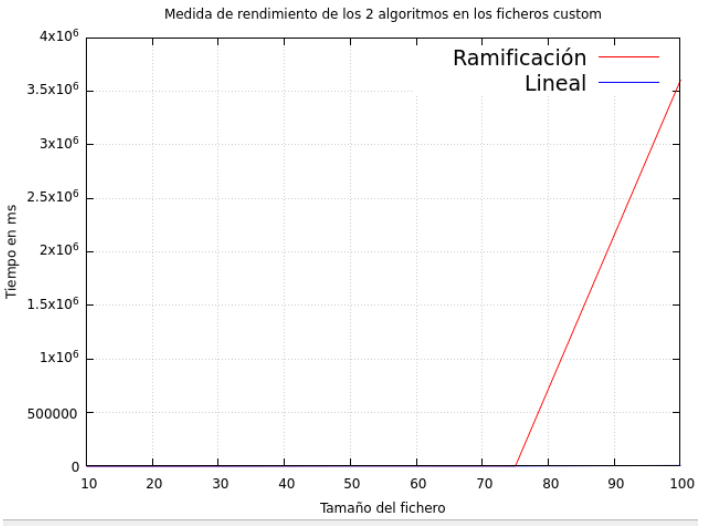
En todas las pruebas realizadas ambas implementaciones alcanzan la misma solución.

En lo referente al rendimiento se han hecho las pruebas de tamaños aleatorios limitados en su máximo por 10,25,50,75,100 estos tamaños aleatorios se aplican tanto en el valor de n como en el de m, para el valor p se emplea el doble de capacidad de forma que la correcta implementación de los algoritmos cobra mucha importancia dada la cantidad de posibles combinaciones.

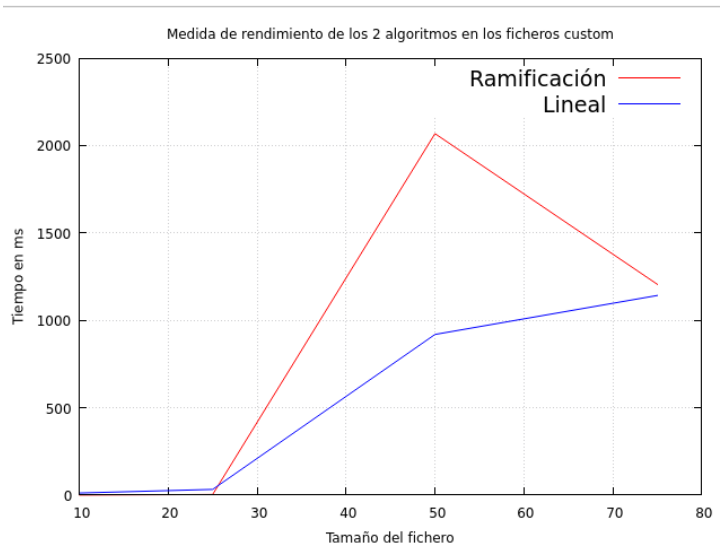
Se han obtenido los siguientes resultados:

```
Ejecutando pruebas estándar
Ejecutando pruebas custom de tamaño 10
La solución en ambos ficheros para tamaño 10 es 79.0
Ejecutando pruebas custom de tamaño 25
La solución en ambos ficheros para tamaño 25 es 133.0
Ejecutando pruebas custom de tamaño 50
La solución en ambos ficheros para tamaño 50 es 3287.0
Ejecutando pruebas custom de tamaño 75
La solución en ambos ficheros para tamaño 75 es 6403.0
Ejecutando pruebas custom de tamaño 85
La solución en ambos ficheros para tamaño 85 es 7496.0
```

Se han obtenido las siguientes métricas de rendimiento:



La solución de ramificación y poda realizada en golang no ha llegado a finalizar su ejecución en un tiempo de 1 hora, es por esto que se detuvo la ejecución dado que eventualmente habría acabado y la diferencia entre el rendimiento de esta solución y la solución lineal resultaba abismal. Es por esto que a continuación tenemos la misma gráfica eliminando este último valor en el que se aprecia la abismal diferencia en cuanto a cómo crece el coste de solucionar el problema conforme a la dificultad del problema base en sí, dado que en programación lineal apenas tardaba 5 segundos en finalizar.



A pesar de que la solución lineal está hecha en Python que como lenguaje es infinitamente más lento obtenemos resultados mejores que la solución de ramificación en Golang.
Para problemas “simples” ambas soluciones son buenas y es relativamente más sencillo plantear una solución de ramificación y poda simple que plantear todas las ecuaciones necesarias para resolver el problema con programación lineal no obstante como queda reflejado en esta prueba a partir de cierta dificultad el problema resulta intratable por el primer método y tenemos que recurrir sí o sí a la programación lineal que es mucho más efectiva.

ANEXO I

```go
package main

import (
    "bufio"
    "fmt"
    "os"
    "sort"
    "strconv"
    "strings"
    "time"
)

type Pedido struct {
    Passenger   int
    StartPos    int
    EndPos      int
    TicketValue int
}

// Pedidos contiene todos los pedidos restantes
// Capacidad disp contiene la capacidad total (capacidad * m) menos la capacidad real usada(suma del vector)
func estimatedMaxIncome(pedidos []Pedido, capacidadDisp int) int {
    aux := capacidadDisp
    for _, pedido := range pedidos {
        if (aux - pedido.TicketValue) > 0 {
            aux -= pedido.TicketValue
        }
    }

    return capacidadDisp - aux
}

// Máximo global para realizar la poda
var absolutMax int

func recursiveSearchDFS(pedidos []Pedido, capacidad []int, capacidadDisp int, alreadyTaken []Pedido, maxIncome int, n
int, m int) (int, []Pedido) {

    if maxIncome > absolutMax {
        absolutMax = maxIncome
    }

    if len(pedidos) == 0 { //Hemos alcanzado una hoja del árbol
        return maxIncome, alreadyTaken
    }

    bestIncome := 0
    bestTaken := make([]Pedido, 0)

    //PODA
    estimatedMax := estimatedMaxIncome(pedidos, capacidadDisp) + maxIncome
    if estimatedMax < absolutMax {
        // fmt.Println("Se ha podado con pedidos:", pedidos, alreadyTaken, " los maximos total y estimados eran:",
absolutMax, estimatedMax)
        return maxIncome, alreadyTaken
    }

    newCapacidad := make([]int, len(capacidad))
    copy(newCapacidad, capacidad)
    admisible := true
```

```
    for i := pedidos[0].StartPos; i < pedidos[0].EndPos; i++ {
        if (newCapacidad[i] + pedidos[0].Passenger) <= n {
            newCapacidad[i] += pedidos[0].Passenger
        } else {
            admisible = false
        }
    }

    if admisible {
        // la solución es válida añadiendo el nuevo pedido
        // fmt.Println("sol admisible", alreadyTaken, pedidos[0])
        newCapacidadDisp := capacidadDisp - pedidos[0].TicketValue
        newMaxIncome := maxIncome
        if maxIncome < (n*m - newCapacidadDisp) {
            newMaxIncome = n*m - newCapacidadDisp
        }
        bestIncome, bestTaken = recursiveSearchDFS(pedidos[1:], newCapacidad, newCapacidadDisp, append(alreadyTaken,
        pedidos[0]), newMaxIncome, n, m)
    } else {
        // fmt.Println("sol NO admisible", alreadyTaken, pedidos[0], capacidad)
    }

    bestIncome2, bestTaken2 := recursiveSearchDFS(pedidos[1:], capacidad, capacidadDisp, alreadyTaken, maxIncome, n,
    m)

    if bestIncome > bestIncome2 {
        return bestIncome, bestTaken
    } else {
        return bestIncome2, bestTaken2
    }
}

func solveProblemInstance(enunciado string, pedidosEnunciado []string) (int, float64) {
    // Extraemos los datos del enunciado
    n, _ := strconv.Atoi(strings.Split(enunciado, " ")[0])
    m, _ := strconv.Atoi(strings.Split(enunciado, " ")[1])

    // Vector que almacena el estado de la capacidad en cada punto del recorrido
    capacidad := make([]int, m)
    // Vector con la información de pedidos ya en formato entero
    pedidos := make([]Pedido, len(pedidosEnunciado))
    for i, pedido := range pedidosEnunciado {
        data := strings.Split(pedido, " ")
        startpos, _ := strconv.Atoi(data[0])
        endpos, _ := strconv.Atoi(data[1])
        passenger, _ := strconv.Atoi(data[2])
        ticketvalue := (passenger) * (endpos - startpos)
        pedidos[i] = Pedido{Passenger: passenger, StartPos: startpos, EndPos: endpos, TicketValue: ticketvalue}
    }

    // Ordenamos el vector de pedidos(para agilizar el precalculo de ganancia)
    sort.Slice(pedidos, func(i, j int) bool {
        return pedidos[i].TicketValue > pedidos[j].TicketValue
    })
    // fmt.Println(pedidos)
    // income := estimatedMaxIncome(pedidos, n*m)
    // fmt.Println("max initial income following the estimation:", income)
    start := time.Now()
    bestSol, _ := recursiveSearchDFS(pedidos, capacidad, n*m, make([]Pedido, 0), 0, n, m)
    elapsed := float64(time.Since(start).Nanoseconds()) / 1000000.0
    return bestSol, elapsed
}
```

```go
func main() {
    //Apertura del fichero de datos
    file, err := os.Open(os.Args[1])
    if err != nil {
        fmt.Printf("Error opening file: %v\n", err)
        os.Exit(1)
    }
    defer file.Close()
    enunciados := make([]string, 0)
    pedidosEnunciados := make([][]string, 0)
    //Lectura de la frase del fichero
    scanner := bufio.NewScanner(file)
    aux := 0
    scanner.Scan()
    linea := scanner.Text()
    for linea != "0 0 0" {
        // fmt.Println(linea)
        enunciados = append(enunciados, linea)
        datos := strings.Split(enunciados[aux], " ")
        p, _ := strconv.Atoi(datos[2])
        // fmt.Printf("p:%d\n", p)
        pedidosEnunciados = append(pedidosEnunciados, make([]string, p))
        for i := 0; i < p; i++ {
            scanner.Scan()
            pedidosEnunciados[aux][i] = scanner.Text()
            // fmt.Println("\t" + pedidosEnunciados[aux][i])
        }
        scanner.Scan()
        linea = scanner.Text()
        aux += 1
    }

    file2 := ""
    if len(os.Args) > 2 {
        file2 = os.Args[2]
    }

    f, err := os.Create("outputRamif" + file2 + ".txt")
    if err != nil {
        fmt.Printf("Error opening file: %v\n", err)
        os.Exit(1)
    }

    defer f.Close()
    w := bufio.NewWriter(f)

    for i := 0; i < aux; i++ {
        absolutMax = 0
        bestSol, elapsed := solveProblemInstance(enunciados[i], pedidosEnunciados[i])
        w.WriteString(strconv.Itoa(bestSol) + ".0 " + strconv.FormatFloat(elapsed, 'f', -1, 64) + "\n")
    }

    w.Flush()
}
```

ANEXO II

```
from mip import Model, xsum, maximize, BINARY
import os
import sys
import time

def cargar_datos(filename):
    enunciados = []
    pedidosEnunciados = []

    with open(filename, 'r') as f:
        for linea in f:
            linea = linea.strip()
            if linea == '0 0 0':
                break

            enunciados.append(linea)
            datos = linea.split(' ')
            p = int(datos[2])
            pedidos = []
            for _ in range(p):
                pedido = f.readline().strip()
                pedidos.append(pedido)
            pedidosEnunciados.append(pedidos)

    return enunciados, pedidosEnunciados

def solve_problem(capacidad, m, p, pedidos):
    # Creación modelo, variables booleanas que representan sin un pedido se ha recogido o no
    # y variables enteras que representan el estado de la capacidad en cada una de las estaciones
    model = Model()
    y = [model.add_var(var_type=BINARY) for _ in range(p)]
    x = [model.add_var(lb=0) for _ in range(m + 1)]

    # Función a maximizar, en este caso el beneficio de los pedidos recogidos
    model.objective = maximize(xsum(pedidos[i][2] * (pedidos[i][1] - pedidos[i][0]) * y[i] for i in
    range(p)))

    # Restricciones
    # El valor de X en cada estación debe ser exactamente igual a la suma de los pasajeros de los pedidos
    recogidos que transiten en esa estación
    # (excluimos los que finalizan dado que los pasajeros bajan al llegar al destino)
    for j in range(m + 1):
        model.add_constr(x[j] == xsum(pedidos[i][2] * y[i] for i in range(p) if pedidos[i][0] <= j <
        pedidos[i][1]))
    # En ningún punto del recorrido el valor local de la X(capacidad) puede superar la capacidad absoluta
    máxima del tren
    for j in range(m + 1):
        model.add_constr(x[j] <= capacidad)

    # Ejecutamos el modelo midiendo el tiempo de ejecución
    start_time = time.time()
    model.optimize()
    end_time = time.time()

    # Extraemos resultados
    max_income = model.objective_value
```

```
    runtime = (end_time - start_time) * 1000

    # Para visualizar mas información
    # x_values = [v.x for v in x]
    # y_values = [v.x for v in y]
    # print(x_values,y_values)

    return max_income, runtime

def solve_problem_instance(enunciado, pedidosEnunciado):

    # Parseamos los datos leidos del fichero correspondientes a una instancia del problema
    capacidad = int(enunciado.split(' ')[0])
    m = int(enunciado.split(' ')[1])
    pedidos = []
    for pedido in pedidosEnunciado:
        data = pedido.split(' ')
        startpos = int(data[0])
        endpos = int(data[1])
        passenger = int(data[2])
        # ticketvalue = passenger * (endpos - startpos)
        pedidos.append((startpos, endpos, passenger))

    # Resolvemos el problema
    max_income, runtime = solve_problem(capacidad, m, len(pedidos), pedidos)

    # Para visualizar mas información
    # Print the solution
    # print(f"Max Income: {max_income}")
    # print(f"Runtime: {runtime} ms")

    return max_income, runtime

if len(sys.argv) > 2:
    SIZE = sys.argv[2]
else:
    SIZE = ""

enunciados, pedidosEnunciados = cargar_datos(sys.argv[1])
with open("outputLinear"+SIZE+".txt", 'w') as f:
    for i in range(len(enunciados)):
        max_income, runtime = solve_problem_instance(enunciados[i],pedidosEnunciados[i])
        f.write(f"{max_income} {runtime}\n")
```

ANEXO III

```ruby
#!/usr/bin/ruby
def generate_input_file(filename,size)
    File.open("#{filename}#{size}.txt", "w") do |file|
        ntest = 1#se podría usar otro valor pero de esta forma podemos comparar los tiempos
        hsize = size / 2
        ntest.times do
            n = rand(size..(size * 2))
            m = rand(hsize..size)
            p = rand(hsize..(size * 2))
            file.puts "#{n} #{m} #{p}"
            p.times do
                s = rand(1..m-1)
                e = rand(s+1..m)
                j = rand(1..n)
                file.puts "#{s} #{e} #{j}"
            end
        end
        file.puts "0 0 0"
    end
end

generate_input_file("tests/input",10)
generate_input_file("tests/input",25)
generate_input_file("tests/input",50)
generate_input_file("tests/input",75)
generate_input_file("tests/input",100)
```