

**Trabajo Práctico Obligatorio**

**Programación III - 2C 2024 - Grupo 1**

Profesor/es:

Cuadrado Estrebou, Maria Fernanda

Foglino, Alejandro Luis

Alumnos:

Viale, Matias – LU: 1108567

Santoro, Axel– LU: 1146936

Santoro, Axel Nicolas– LU: 1146936

Ghisoni, Francisco– LU: 1117937

Rodríguez, Roxana– LU: 1164643

Grasso Lavrivera, Juan Francisco-LU: 1123151

Buenos Aires, 08 de Noviembre de 2024.-

**Tabla de Contenidos**

[Introducción 3](#_30j0zll)

[Descripción del Problema 3](#_1fob9te)

[Estrategia de Resolución 3](#_3znysh7)

[Pseudocódigo del Algoritmo de Resolución del Problema 3](#_2et92p0)

[Análisis de Complejidad Temporal](#_tyjcwt) 6

[Conclusiones 3](#_3dy6vkm)

[Bibliografía 4](#_1t3h5sf)

# Introducción

*El presente trabajo describe la utilización de la técnica de backtracking para determinar todas las blockchain que cumplan con un conjunto amplio y complejo de reglas. En este trabajo una blockchain es una cadena de bloques que contienen dentro transacciones, cada bloque está sujeto a condiciones que deben cumplir para ser válidos como por ejemplo que su valor no supere un cierto valor o su tamaño no sea superior a otro valor. Consideramos una blockchain válida la que cumpla con las reglas de transacciones y bloques y llegue a la cantidad de bloques solicitados en los cuales no puedan agregarse más transacciones o nos quedemos sin transacciones a agregar, siempre considerando el tope máximo de bloques.*

# Descripción del Problema

## Estrategia de Resolución

*Para determinar las distintas blockchain válidas utilizaremos la técnica de Backtracking. Está técnica nos permite evaluar, en forma recursiva, si los bloques son válidos e ir generando una cadena de bloques (blockchain) hasta llegar al máximo de bloques permitidos.   
Si se llega al máximo de bloques permitidos y no existen más transacciones que agregar consideramos está cadena como válida   
Para determinar la solución tendremos una lista de transacciones y las diferentes condiciones que deben cumplir los bloques. Además cada transacción contiene sus dependencias y firmas necesarias para poder ser utilizada.*

*La estrategia en cuestión consiste en evaluar las diferentes combinaciones de transacciones, se irán tomando de a una cada transacción en cada paso recursivo y se verificará su validez, es decir si, las diferentes reglas existentes y si puede ingresar al bloque actual o ingresar al siguiente bloque.  
Se deberá explorar toda combinación de transacciones posibles. En caso de no ser válida, se cortará el backtracking en ese punto volviendo al paso anterior.*

## Pseudocódigo del Algoritmo de Resolución del Problema

**Algoritmo blockchain**

**Entrada:** indice : int,

transacciones : Vector<Transacción>,

bloquesActuales : Vector<Bloque>,

soluciones : Vector<Vector<Bloque>>,

maxTamanioBloque : int,

maxValorBloque : int,

maxTransacciones : int,

maxBloque : int

soluciones ← inicializar()

bloques ← inicializar()

backtracking(0, transacciones, bloques, soluciones, maxTamanioBloque, maxValorBloque, maxTransacciones, maxBloques)

**devolver** soluciones

**Algoritmo backtracking**

**Entrada:** transacciones : Vector<Transacción>,

bloquesActuales : Vector<Bloque>,

soluciones : Vector<Vector<Bloque>>,

maxTamanioBloque : int,

maxValorBloque : int,

maxTransacciones : int,

maxBloque : int

**SI** ( bloquesActuales.longitud() <= maxBloques **AND** transacciones.longitud() = 0) {

soluciones.agregar(bloquesActuales);

devolver;

}**SINO** {

devolver;

}

**PARA** i **DESDE** 0 **HASTA** transacciones.longitud() -1**{**

transaccionSeleccionada <- transacciones.get(i)

**SI** ( bloquesActuales.longitud() = 0) {

bloqueActual <- crearBloque()

estaAgregado <- false

}

**SINO {**

bloqueActual <- bloquesActuales.obtener(bloquesActuales.longitud() -1)

estaAgregado <- true

**SI** (!puedeAgregarTransaccion(

bloqueActual, transaccionSeleccionada, maxTamanioBloque,

maxValorBloque,

maxTransacciones)

**{**

bloqueActual <- crearBloque()

estaAgregado <- false

**}**

}

**SI** (puedeAgregarTransaccion(

bloqueActual, transaccionSeleccionada, maxTamanioBloque,

maxValorBloque,

maxTransacciones)

**{**

bloqueActual.transacciones.agregar(transaccionSeleccionada)

bloqueActual.valor ← bloqueActual.valor + transaccionSeleccionada.valor

bloqueActual.tamaño <- bloqueActual.tamaño + transaccionSeleccionada.tamaño

**SI**(!estaAgregado)

bloquesActuales.agregar(bloqueActual)

backtracking(

transacciones,

bloquesActuales,

soluciones,

maxTamanioBloque,

maxValorBloque,

maxTransacciones, maxBloques

)

bloqueActual.transacciones.remove(bloqueActual.transacciones.longitud() - 1)

bloqueActual.valor ← bloqueActual.valor - transaccionSeleccionada.valor

bloqueActual.tamaño <- bloqueActual.tamaño - transaccionSeleccionada.tamaño

**}**

**SI** bloqueActual.transacciones.longitud()= 0 {

bloquesActuales.eliminar(bloquesActuales.longitud() - 1)

}

**}**

**Algoritmo puedeAgregarTransaccion**

**Entrada:** bloque : Bloque,

transaccion : Transacción,

maxTamanioBloque : int,

maxValorBloque : int,

maxTransacciones : int,

transaccionesPorChequear: Vector<Transaccion>

**Salida:** boolean

**devolver** !transaccionsPorChequear.contains(dependencia)

**AND** transaccionActual.firmasActuales >= transaccionActual.firmasRequeridas

**AND** (bloque.valorTotal + transaccionActual.valor ) % 10 == 0

**AND** transaccionActual.getvalor + bloque.getValorTotal <= maxValorBloque

**AND** transaccionActual.getTamaño + bloque.getTamañoTotal() <= maxTamañoBloque

**AND** bloque.getTransacciones.longitud() < maxTransacciones;

## Análisis de Complejidad Temporal

***Complejidad Temporal Teórica***

*La llamada recursiva es del tipo T(n-b) donde la función queda determinada por:*

Texto

Descripción generada automáticamente con confianza baja

*Donde “a” es la cantidad de llamadas recursivas en el peor de los casos, “b” es la cantidad de unidades que disminuye la entrada en cada llamado recursivo y “k” es el grado del polinomio p(n) de aquellas sentencias que se ejecutan por fuera del llamado recursivo.*

Texto, Carta

Descripción generada automáticamente

*En nuestro caso cada nivel es representado por una transacción que no haya sido evaluada, por eso nuestro “a”>1, luego tenemos el “b” que representa la cantidad disminuida por nivel, en nuestro caso “b”=1*

*En conclusión, la complejidad temporal está dado por la siguiente expresión.*

*T(n)=θ(mn), donde m es la cantidad de transacciones.*

***Complejidad Temporal Práctica***

*Pudimos comprobar la complejidad teórica viendo como los valores van escalando a medida que tenemos más transacciones*

| Transacciones simples |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Cantidad | maxTamanio | maxValorBloque | maxTransacciones | maxBloques | Tiempo(segundos) |
| 100000 | 1000 | 1000 | 50 | 100 | 1 |
| 500000 | 1000 | 1000 | 50 | 100 | 31 |
| 900000 | 1000 | 1000 | 50 | 100 | 101 |
| 1500000 | 1000 | 1000 | 50 | 100 | 280 |

| Transacciones mixtas |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Cantidad | maxTamanio | maxValorBloque | maxTransacciones | maxBloques | Tiempo(segundos) |
| 100000 | 1000 | 1000 | 50 | 100 | 2 |
| 500000 | 1000 | 1000 | 50 | 100 | 35 |
| 900000 | 1000 | 1000 | 50 | 100 | 107 |
| 1500000 | 1000 | 1000 | 50 | 100 | 292 |

# 

# Conclusiones

*A partir de la implementación de la técnica de Backtracking para resolver el problema propuesto, hemos podido corroborar empíricamente la complejidad temporal obtenida en forma teórica.* *Sin embargo, por el momento, carecemos de la poda lo que nos impidió comprobar la complejidad práctica. (Se adjunta la implementación del código para poda, para su revisión, queda pendiente realizar las pruebas prácticas)*

# 