FACULTAD DE INGENIERÍA Y CIENCIAS EXACTAS

DEPARTAMENTO DE TECNOLOGÍA INFORMÁTICA

TPO: Backtracking

Profesor/es:

Cuadrado Estrebou, Maria Fernanda

Foglino, Alejandro Luis

Grupo:

Koller, Jose Luis – LU: 1119528

Casañas, Juan Bautista – LU: 1161617

Fernandez, Nicolás - LU: 1148903

Gutik, Nicolás - LU: 1155519

Cassarino, Máximo - LU: 1158725

Buenos Aires, 08 de Noviembre de 2024.-

Tabla de Contenidos

Introducción

El presente trabajo práctico se centra en la implementación de un algoritmo de backtracking para la construcción de una blockchain. La blockchain se caracteriza por garantizar la integridad de las transacciones mediante reglas criptográficas y consensos entre los nodos participantes.

El desafío principal consiste en diseñar e implementar un algoritmo que sea capaz de construir una cadena de bloques válida, considerando múltiples restricciones y reglas de validación. Y se debe de tener en cuenta cosas como:

- Gestión de transacciones interdependientes
- Validación de firmas múltiples
- Implementación de restricciones de prueba de trabajo
- Manejo de prioridades en las transacciones
- Control de tamaño máximo de bloques
- Validación de valores máximos por bloque

El algoritmo debe ser capaz de explorar exhaustivamente todas las combinaciones posibles de bloques que cumplan con las restricciones establecidas. Este enfoque permite no solo encontrar una solución viable, sino identificar todas las posibles configuraciones de la blockchain que satisfagan los requisitos especificados.

La implementación debe considerar aspectos críticos como la eficiencia computacional, la correcta validación de las reglas de negocio y la gestión adecuada de las dependencias entre transacciones, todo ello manteniendo la integridad y coherencia propias de una blockchain.

Descripción del Problema

El problema consiste en desarrollar un algoritmo que permita construir una blockchain válida a partir de un conjunto de transacciones, respetando múltiples restricciones y reglas de validación. El sistema debe ser capaz de encontrar todas las combinaciones posibles de bloques que cumplan con los criterios establecidos.

Componentes del Sistema

Transacciones

Cada transacción en el sistema está caracterizada por los siguientes atributos:

- **Tamaño**: Medido en KB, representa el espacio que ocupa la transacción
- **Valor**: Expresado en satoshis, indica el monto de la transacción
- **Dependencias**: Conjunto de transacciones que deben ser procesadas previamente
- **Firmas**: Cantidad de firmas requeridas para validar la transacción

Bloques

Los bloques que conforman la blockchain deben cumplir con las siguientes restricciones:

1. **Tamaño Máximo**: 1 MB por bloque
2. **Cantidad de Transacciones**: Máximo 3 transacciones por bloque
3. **Valor Total**: La suma de los valores de las transacciones no debe superar 100 satoshis
4. **Prueba de Trabajo (PoW)**: La suma de los valores de las transacciones debe ser divisible por 10

Reglas de Validación

1. **Dependencias**
    - Una transacción solo puede incluirse si todas sus dependencias han sido procesadas en bloques anteriores
    - Las dependencias forman un grafo dirigido que no debe contener ciclos
2. **Firmas Múltiples**
    - Cada transacción requiere un número específico de firmas para ser válida
    - Las firmas deben verificarse antes de incluir la transacción en un bloque

Entrada del Sistema

El sistema recibe como entrada:

- Lista de transacciones con sus respectivos atributos
- Tamaño máximo permitido por bloque
- Número máximo de bloques
- Cantidad de firmas requeridas

Salida Esperada

El algoritmo debe producir:

- Todas las posibles combinaciones válidas de bloques que cumplan con las restricciones
- Cada solución debe especificar la distribución de transacciones en bloques
- Las soluciones deben respetar todas las reglas de validación establecidas

Esta descripción del problema establece el marco para el desarrollo de una solución basada en backtracking, que debe explorar el espacio de soluciones mientras válida las restricciones.

Estrategia de Resolución

Tal como se mencionó previamente, se utilizara la técnica de backtracking, la cual nos va a permitir todas las soluciones posibles, construyéndolas incrementalmente y descartando las que no satisfagan, se intentara realizar poda temprana siempre que se pueda y las restricciones establecidas.

Fundamentos de la Estrategia

Para la estrategia del ejercicio, en el cual estaremos utilizando backtracking, vamos a verlo desde el lado del árbol de decisión, estableciendo sus condiciones de poda, validaciones y la implementación del backtracking.

Como condiciones sabemos que, las transacciones deben de tener sus dependencias resueltas previamente, la firmas requeridas tienen que ser menor o igual a las disponibles, el valor debe mantenerse de 100 satoshis o menor y el tamaño total debe ser 1MB o menos.

1. Estructura del Árbol de Decisión

Primer Nivel

- Selección de la primera transacción al iniciar un bloque
- Representar el estado inicial de cada bloque en construcción
- Se evalúa el tamaño y valor inicial

Nivel Intermedio

- Se adicionan transacciones al bloque actual
- Ramificaciones considerando distintas combinaciones de transacciones
- Acumulación de tamaño y valor

Nivel Final

- Representación del bloque completo
- Validación del bloque final
- Construcción del siguiente bloque si quedan transacciones pendientes

2. Condición de Poda

Poda por Dependencias

- Se verifica antes de agregar una transacción
- Una transacción solo puede agregarse si todas sus dependencias están en bloques anteriores
- Ejemplo en árbol: "Poda: T2 dep T1" y "Poda: T4 dep T3"

Poda por Valor

- Se verifica después de agregar cada transacción
- El valor total del bloque no puede exceder 100 satoshis
- Ejemplo en árbol: "Poda: >100s"

Poda por Tamaño

- Se verifica el tamaño acumulado del bloque
- No debe exceder 1MB (1000KB)
- Se calcula sumando los tamaños de las transacciones

3. Amplitud

1. **Primera Transacción del Bloque**
    - Se consideran todas las transacciones sin dependencias pendientes
    - Se priorizan transacciones con menor número de dependencias
2. **Transacciones Subsiguientes**
    - Se evalúan todas las transacciones restantes no utilizadas
    - Se verifican restricciones incrementales:
        - Tamaño acumulado
        - Valor acumulado
        - Dependencias resueltas

Control de Estado

- Mantener registro de:
    - Transacciones utilizadas
    - Dependencias resueltas
    - Valores acumulados por bloque
    - Tamaños acumulados por bloque

4. Validación

Por Transacción

- Verificar dependencias resueltas
- Calcular nuevo tamaño total
- Calcular nuevo valor total
- Verificar número de firmas requeridas

Por Bloque

- Máximo 3 transacciones
- Valor total ≤ 100 satoshis
- Suma de valores divisible por 10 (prueba de trabajo)
- Tamaño total ≤ 1MB

5. Backtracking

Selección

- Elegir próxima transacción candidata
- Verificar dependencias resueltas
- Calcular impacto en el bloque actual

Validación

- Aplicar criterios de validación incremental
- Verificar condiciones de poda
- Evaluar si el bloque está completo

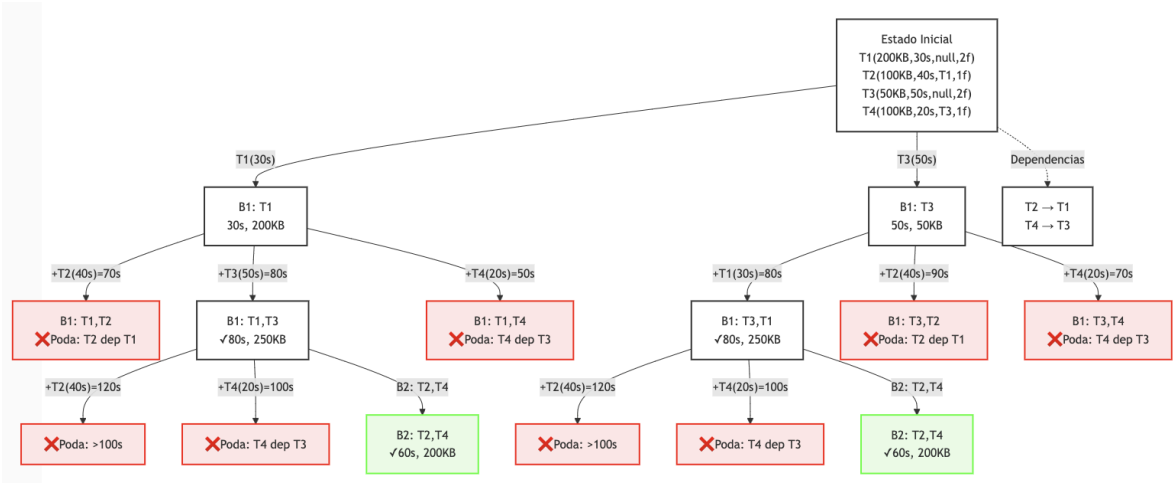
Decisión

- Si es válido: continuar con siguiente transacción
- Si no es válido: retroceder y probar siguiente opción
- Si bloque completo: iniciar nuevo bloque

Retroceso

- Restaurar estado anterior
- Marcar camino como explorado
- Seleccionar siguiente alternativa disponible

Ejemplo



Estado Inicial

Transacciones Disponibles:

- T1: 200KB, 30s, sin dependencias, 2 firmas
- T2: 100KB, 40s, depende de T1, 1 firma
- T3: 50KB, 50s, sin dependencias, 2 firmas
- T4: 100KB, 20s, depende de T3, 1 firma

Exploración del Árbol - Rama Izquierda

1. Inicio con T1 Bloque 1: T1
- Tamaño: 200KB
- Valor: 30s
- Estado: Válido ✓
- Puede continuar expandiéndose
2. Intentos de expansión desde T1

a) Intento: T1 + T2 Bloque 1: T1, T2

- Valor acumulado: 70s
- Poda: T2 depende de T1 ✗
- Razón: T2 no puede estar en el mismo bloque que T1

b) Intento: T1 + T3 Bloque 1: T1, T3

- Tamaño: 250KB
- Valor: 80s
- Estado: Válido ✓ (divisible por 10)

c) Intento: T1 + T4 Bloque 1: T1, T4

- Poda: T4 depende de T3 ✗
- Razón: T3 no está en bloques anteriores

3. Expansión desde T1, T3

a) Intento: T1, T3 + T2 Bloque 1: T1, T3, T2

- Valor acumulado: 120s
- Poda: Excede 100s ✗

b) Intento: T1, T3 + T4 Bloque 1: T1, T3, T4

- Poda: T4 depende de T3 ✗
- Razón: T4 no puede estar en el mismo bloque que T3

Exploración del Árbol - Rama Derecha

1. Inicio con T3 Bloque 1: T3

- Tamaño: 50KB
- Valor: 50s
- Estado: Válido ✓

2. Intentos de expansión desde T3

a) Intento: T3 + T1 Bloque 1: T3, T1

- Tamaño: 250KB
- Valor: 80s
- Estado: Válido ✓

b) Intento: T3 + T2 Bloque 1: T3, T2

- Poda: T2 depende de T1 ✗
- Razón: T1 no está en bloques anteriores

c) Intento: T3 + T4 Bloque 1: T3, T4

- Poda: T4 depende de T3 ✗
- Razón: T4 no puede estar en el mismo bloque que T3

Formación de Bloques Válidos

Solución 1 Bloque 1: T1, T3

- Tamaño: 250KB
- Valor: 80s
- Estado: Válido ✓

Bloque 2: T2, T4

- Tamaño: 200KB
- Valor: 60s
- Estado: Válido ✓
- Dependencias satisfechas (T1 y T3 están en B1)

Solución 2 Bloque 1: T3, T1

- Tamaño: 250KB
- Valor: 80s
- Estado: Válido ✓

Bloque 2: T2, T4

- Tamaño: 200KB
- Valor: 60s
- Estado: Válido ✓
- Dependencias satisfechas (T1 y T3 están en B1)

Pseudocódigo del Algoritmo de Resolución del Problema

FUNCIÓN construirBloqueBlockchain(transacciones, maxTamañoBloque, maxValorBloque,maxBloques,maxTransacciones, numFirmasRequeridas)

```
 // Validación de parámetros de entrada
   SI transacciones está vacía ENTONCES
      RETORNAR lista vacía
   FIN SI

   SI maxTamañoBloque <= 0 O maxBloques <= 0 O numFirmasRequeridas < 0
ENTONCES
      LANZAR Error("Parámetros inválidos")
   FIN SI

   // Estructuras de datos necesarias
   listaSoluciones = nueva Lista()

   PARA i = 0 HASTA longitud(transacciones) - 1
                actSolucion[i] ← 0
   FIN PARA
```

```
totalValorBloque ← 0
totalTamañoBloque ← 0
totalCantidadTransacciones ← 0
etapa ← 0

backtrackBlockchain(transacciones,actSolucion,totalValorBloque,
totalTamañoBloque,totalCantidadTransacciones,etapa,listaSoluciones,maxTamañoBloque,
maxValorBloque,maxBloques,maxTransacciones)

FUNCION backtrackBlockchain(transacciones,actcSolucion,totalValorBloque,
totalTamañoBloque,totalCantidadTransacciones,etapa,listaSoluciones,maxTamañoBloque,
maxValorBloque,maxBloques,maxTransacciones)
   PARA i = 0 HASTA 1
     actSolucion[j] ← i
     totalValorBloque ← totalValorBloque + transacciones[etapa].i
     SI etapa == longitud(transanciones)
       SI longitud(listaSoluciones) <= maxBloques
          SI totalCantidadTransacciones <= maxTransacciones
             SI transacciones[etapa].dependencia != null
                SI totalTamañoBloque <= maxTamañoBloque
                   SI totalValorBloque <= maxValorBloque
                      PARA n=0 HASTA transaccion[etapa].firmas
                        transaccion[etapa].consumirFirma()
                      FIN PARA
                      listaSoluciones.agregar(actSolucion)
                      totalCantidadTransacciones ←  totalCantidadTransacciones + 1
                   FIN SI
                FIN SI
             SINO
               SI existeDependenciaResuelta(transacciones[etapa].dependencias)
```

```
                SI totalTamañoBloque <= maxTamañoBloque
                  SI totalValorBloque <= maxValorBloque
                    PARA n=0 HASTA transaccion[etapa].firmas
                      transaccion[etapa].consumirFirma()
                    FIN PARA
                    listaSoluciones.agregar(actSolucion)
                    totalCantidadTransacciones ← totalCantidadTransacciones + 1
                  FIN SI
                FIN SI
              FIN SI
            FIN SI
          SINO
            SI totalTamañoBloque <= maxTamañoBloque
              SI totalValorBloque <= maxValorBloque
backtrackBlockchain(transacciones,actSolucion,totalValorBloque,totalTamañoBloque,totalC
antidadTransacciones,etapa+1,listaSoluciones,maxTamañoBloque,
maxValorBloque,maxBloques,maxTransacciones)
              FIN SI
            FIN SI
        FIN SI
        FIN PARA
        FIN  backtrackBlockchain
```

```
FUNCION existeDependenciaResuelta(transaccionDependencia, listaSolucion)
        PARA i=0 HASTA longitud(listaSolucion)
                SI listaSolucion[i][transaccionDependencia] == 1
                        RETORNAR VERDADERO
                FIN SI
        FIN PARA
        RETORNAR FALSO
DEVOLVER listaSoluciones
FIN construirBloqueBlockchain
```

Análisis de Complejidad Temporal

El backtracking es una técnica exhaustiva que explora todas las posibles soluciones, pensándolo en este contexto de blockchain, se va a determinar por la cantidad de transacciones y sus restricciones.

Cuando hablamos de las restricciones tenemos que tener en cuenta cosas como la cantidad de firmas requeridas por transacciones, la dependencia entre ellas y los límites de tamaño y costo.

Complejidad Temporal Teórica

Si analizamos el algoritmo podemos darnos cuenta que posee una forma similar al divide y conquista. Si lo analizamos como tal, nos damos cuenta que se trata de un caso de sustracción ya que en cada llamada recursiva se reduce en un elemento, y teniendo en cuenta su fórmula :

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n-b) + p(n) & \text{si } n \geq b \end{cases}$$

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \; div \; b}) & \text{si } a > 1 \end{cases}$$

Toda la complejidad temporal se encuentra en la función backtrackBlockchain, ya que lo demás son llamados a funciones, inicializaciones, asignaciones,etc.

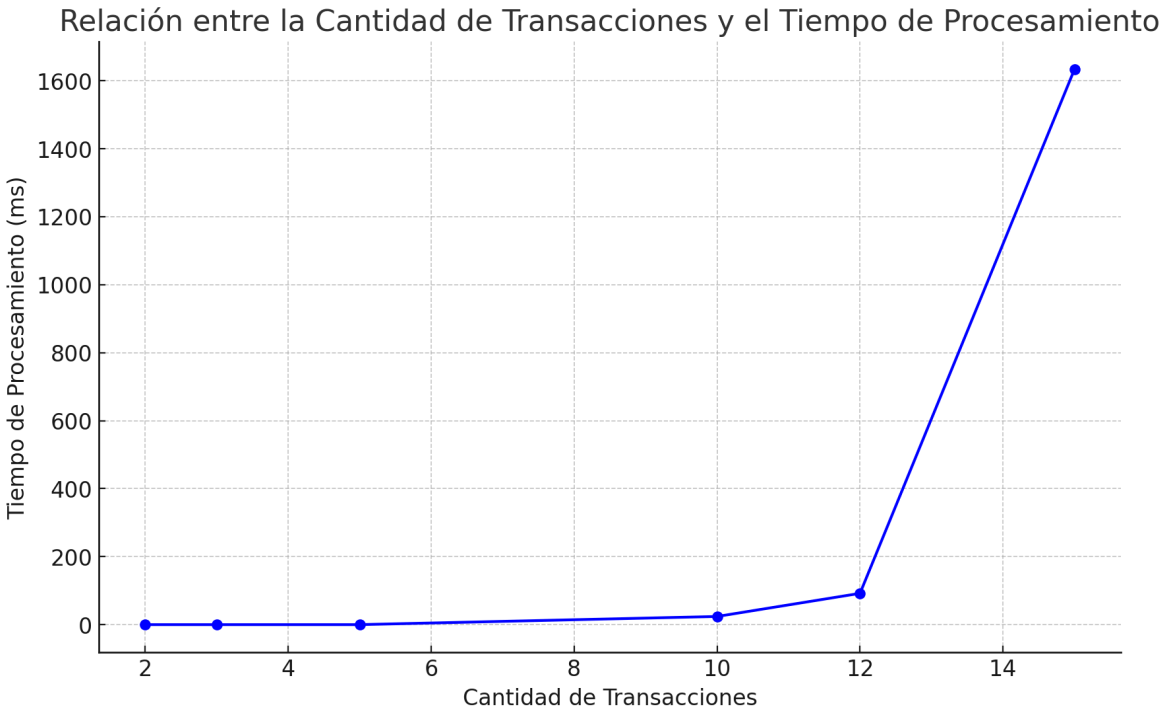
Identificamos a A, B y K y nos queda lo siguiente:

A = 2 (ya que en el peor de los casos se itera dos veces el ciclo)

B = 1 (es el número que resta)

K = 1 (es el grado del polinomio de las funciones que se utilizan y son externas al algoritmo, como podemos ver se utiliza la función existeDependenciaResuelta y es de orden O(n) por el ciclo que realiza). Resumiendo: $O(2^{n \; div \; b})$, es decir que la complejidad temporal teórica del algoritmo es de $O(2^n)$

Complejidad Temporal Práctica



Como podemos ver en el gráfico después de las 10 transacciones la complejidad temporal se dispara exponencialmente, hasta tal punto que con más de 15 transacciones se cuelga el programa. Probamos con 20 y se nos colgó todas las veces que intentamos, este es un claro ejemplo de que se deberían buscar formas de mejorarlo, para poder utilizar el algoritmo de forma eficiente.

Conclusiones

El presente trabajo práctico se centró en el desarrollo e implementación de un algoritmo de backtracking para la construcción de una blockchain. Durante su desarrollo, nos enfrentamos a diversos desafíos que nos permitieron crecer tanto en el aspecto técnico como en nuestra capacidad de análisis y resolución de problemas.

Al iniciar el desarrollo del pseudocódigo, nos encontramos con algunas dificultades debido a la falta de familiaridad con este tipo de trabajo. Sin embargo, este proceso inicial resultó ser fundamental, ya que al momento de realizar la implementación en código, pudimos identificar simplificaciones y mejoras que no eran evidentes en un principio. La implementación en Java nos permitió lograr una estructura modular y mantenible, separando adecuadamente las responsabilidades en las distintas clases (Transaccion, Bloque, AlgoritmoDeBlockchain), lo que facilitará futuras modificaciones y extensiones del sistema.

En cuanto al análisis de complejidad, inicialmente determinamos que la complejidad teórica era O(2^n) , donde n representa el número de transacciones. Sin embargo, una vez implementado el código y realizado un análisis más detallado, observamos que la complejidad temporal quedaba expresada como O(n^m), siendo n la cantidad de ramificaciones y m la profundidad del árbol. Esta diferencia entre el análisis inicial y el resultado final nos ayudó a comprender mejor el comportamiento real del algoritmo y la importancia de las optimizaciones implementadas.

Si bien la solución desarrollada cumple con los requerimientos establecidos y demuestra ser eficiente para los casos de prueba propuestos, somos conscientes que no es escalable por la alta complejidad temporal que presenta.

Como reflexión final, este trabajo nos permitió comprender la importancia fundamental del diseño cuidadoso de las estructuras de datos y la necesidad de entender en profundidad el dominio del problema antes de comenzar la implementación. Un aprendizaje significativo fue la evolución de nuestro enfoque: inicialmente intentamos resolver todos los problemas en una única función, pero a medida que avanzamos, la modularización del código nos llevó a una solución más simple, comprensible y eficiente.

El proceso completo, desde el análisis inicial hasta la implementación final, ha sido un valioso ejercicio que nos permitió aplicar y profundizar nuestros conocimientos sobre algoritmos, estructuras de datos y técnicas de programación, mientras desarrollábamos una solución práctica para un problema complejo y actual como es la construcción de una blockchain.

Nueva implementación del pseudocódigo:

```
FUNCIÓN construirBloqueBlockchain(transacciones, maxTamañoBloque,
maxValorBloque,maxBloques,maxTransacciones, numFirmasRequeridas)

        // Validación de parámetros de entrada
        SI transacciones está vacía
                ENTONCES RETORNAR lista vacía
        FIN SI

        SI maxTamañoBloque <= 0 O maxValorBloques <= 0
        O maxBloques <=0  O maxTransacciones<=0
        ENTONCES
```

```
            RETORNAR lista vacía
        FIN SI

        // Estructuras de datos necesarias
         listaSoluciones = nueva Lista()
        actSolucion = nueva Lista()

        PARA i = 0 HASTA longitud(transacciones)
                actSolucion[i] ← 0
        FIN PARA

        totalValorBloque ← 0
        totalTamañoBloque ← 0
         totalCantidadTransacciones ← 0
        etapa ← 0

        backtrackBlockchain(transacciones,actSolucion,totalValorBloque,

        totalTamañoBloque,totalCantidadTransacciones,etapa,listaSoluciones,maxTamaño
        Bloque, maxValorBloque,maxBloques,maxTransacciones)

        RETORNAR convetirSolucionesABlockchain(listaSoluciones,transacciones)

FIN construirBloqueBlockchain

FUNCION backtrackBlockchain(transacciones,actSolucion,totalValorBloque,
totalTamañoBloque,totalCantidadTransacciones,etapa,listaSoluciones,maxTamañoBloque,
maxValorBloque,maxBloques,maxTransacciones)

        numBloques ← 0
        PARA j=0 HASTA etapa
                SI actSolucion[i] == 2
                        numBloques ++
                FIN SI
        FIN PARA

        PARA i = 0 HASTA 2

                Si i == 2 y numBloques >= maxBloques -1
```

```
                ENTONCES CONTINUAR
        FIN SI


        actSolucion[etapa] ← i
        nuevoValorBloque ← totalValorBloque + transaccion[etapa] * i
        nuevoTamañoBloque ← totalTamanioBloque + transaccion[etapa] * i
        nuevaCaantidadTransacciones ← totalCantidadTransacciones + i

        SI nuevoValorBloque > maxValorBloque O nuevoValorBloque % 10 != 0 O
         nuevaCantidadTransacciones > maxTransacciones O
nuevoTamañoBloque >
        maxTamanioBloque
        ENTONCES
                CONTINUAR
        FIN SI

        SI i == 1 Y NO ESTÁ FIRMADA
        ENTONCES
                CONTINUAR
        FIN SI


        SI etapa == longitud(transacciones) -1
                SI validarDependencias(transacciones[etapa],
                      actSolucion,transacciones)
                         bloquesSolucion←nueva Lista()
                         nuevaActSolucion←nueva Lista()
                         bloquesSolucion.agregar(nuevaActSolucion)
                         SI longitud(bloquesSolucion) < maxBloques
                                  listaSoluciones.agregar(bloquesSolucion)
                         FIN SI

                FIN SI
        SINO

                backtrackBlockchain(transacciones,actSolucion,nuevoValorBloque,
                nuevoTamañoBloque,nuevaCantidadTransacciones,etapa+1,listaSol
                uciones,maxTamañoBloque,
```

```
                    maxValorBloque,maxBloques,maxTransacciones)

            FINSI

        FINPARA

FIN backtrackBlockchain

FUNCION validarDependencias(transaccion,solucion,transacciones)
        SI transaccion.traerDependencia == NULL
        ENTONCES
            RETORNAR VERDADERO
        FIN SI
        RETORNAR existeDependenciaResuelta(transaccion.traerDependencia
        ,solucion,transacciones)
FIN validarDependencias

FUNCION existeDependenciaResuelta(dependencia,solucion,transacciones)

        PARA i = 0 HASTA longitud(solucion)
            SI solucion[i] == 1 Y dependencia ES IGUAL A transaccion[i]
            ENTONCES
                RETORNAR VERDADERO
            FINSI
        FIN PARA
        RETORNAR FALSO
FIN existeDependenciaResuelta

FUNCION convertirSolucionesABlockchain(soluciones,transacciones)
        blockchain = nueva Lista()

        PARA CADA solucionBloques en soluciones
            chain = nueva Lista()
            blockTransactions= nueva Lista()



            PARA i= 0 HASTA longitud(solucionBloques[0])
                decision ← solucionBloques[0][i]
```

```
                    SI decision > 0
                        blockTransactions.agregar(transacciones[i])
                        SI decision == 2 O i ==longitud(solucionBloques[i])-1
                            Si blockTransactions NO ESTA VACIO
                                chain.agregar(blockTransactions)
                                blockTransactions ← nueva Lista()
                            FIN SI

                        FIN SI
                    FIN SI
                FIN PARA

                SI blockTransactions NO ESTA VACIO
                    chain.agregar(blockTransactions)
                    blockchain.agregar(chain)
                FINSI

        FIN PARA
        RETORNAR blockchain
FIN convertirSolucionesABlockchain
```

Bibliografía

UVA 06 - Aula Virtual UADE

UVA 07 - Aula Virtual UADE