FACULTAD DE INGENIERÍA Y CIENCIAS EXACTAS

DEPARTAMENTO DE TECNOLOGÍA INFORMÁTICA

TPO | Blockchain

Profesor/es:

CUADRADO ESTREBOU, MARIA FERNANDA

FOGLINO, ALEJANDRO LUIS

Grupo:

Musacchio, Ignacio Alfredo – LU: 1150581

Leguizamon, Federico - LU: 1162132

Penalva, Santiago Nicolás – LU: 1182245

Seita, Joaquin - LU: 1165187

Buenos Aires, 08 de Noviembre de 2024.-

Tabla de Contenidos

Introducción 3

Descripción del Problema 3

Estrategia de Resolución 3

Pseudocódigo del Algoritmo de Resolución del Problema 6

Análisis de Complejidad Temporal 10

Conclusiones 11

Introducción

En este trabajo práctico, se plantea la necesidad de implementar un algoritmo que permita construir una blockchain válida a partir de un conjunto de transacciones bajo condiciones específicas y restricciones bien definidas. Las transacciones poseen atributos como el tamaño, el valor de cada transacción, las dependencias entre sí y la necesidad de múltiples firmas. Además, se deben considerar reglas como el límite de tamaño de los bloques y la prueba de trabajo

A partir de esta necesidad, desarrollaremos el pseudocódigo de la solución y también un implementación en Java del mismo. A su vez, analizaremos nuestra propuesta evaluando su complejidad temporal y describiendo la estrategia de resolución.

Descripción del Problema

Estrategia de Resolución

La estrategia utilizada para encontrar todas las combinaciones de blockchains válidas que cumplan con las restricciones definidas se basa en la técnica de backtracking la cual nos permitirá explorar todas las configuraciones posibles de transacciones distribuidas en distintos bloques.
El algoritmo construirá todas las combinaciones posibles de bloques, cada uno de los cuales cumple con las restricciones de tamaño, valor, cantidad de transacciones y dependencias de transacciones. Se inicializará partiendo de una lista vacía de bloques y procesando una por una las transacciones, distribuyendo en los bloques existentes o creando nuevos.
Al estar utilizando backtracking, se realizan llamadas recursivas. En cada nivel de recursión, se decidirá qué hacer con una transacción específica: Por un lado se agregará ésta al bloque actual, siempre y cuando se respete el máximo de tamaño de bloque, el máximo de valor del bloque y el número máximo de transacciones por bloque. Por otro lado se creará un nuevo bloque en el que se añadirá esta transacción, mientras no se exceda el número máximo de bloques. Luego de tomar esta decisión, se procederá a evaluar la siguiente transacción con el estado actual de los bloques.
Con el objetivo de no explorar ramas de decisión que se pueda anticipar que no serán válidas, se implementará también una estrategia de poda. Para ello, antes de agregar una transacción a un bloque o de crear uno nuevo, se verificará que se cumplan las siguientes restricciones:

- Máximo de tamaño de bloque (suma del tamaño de cada transacción del bloque)
- Máximo de valor de bloque (suma del valor de cada transacción del bloque)
- Cantidad máxima de transacciones en el bloque
- Dependencias entre transacciones (que las dependencias de la transacción evaluada ya se encuentren en el bloque actual)

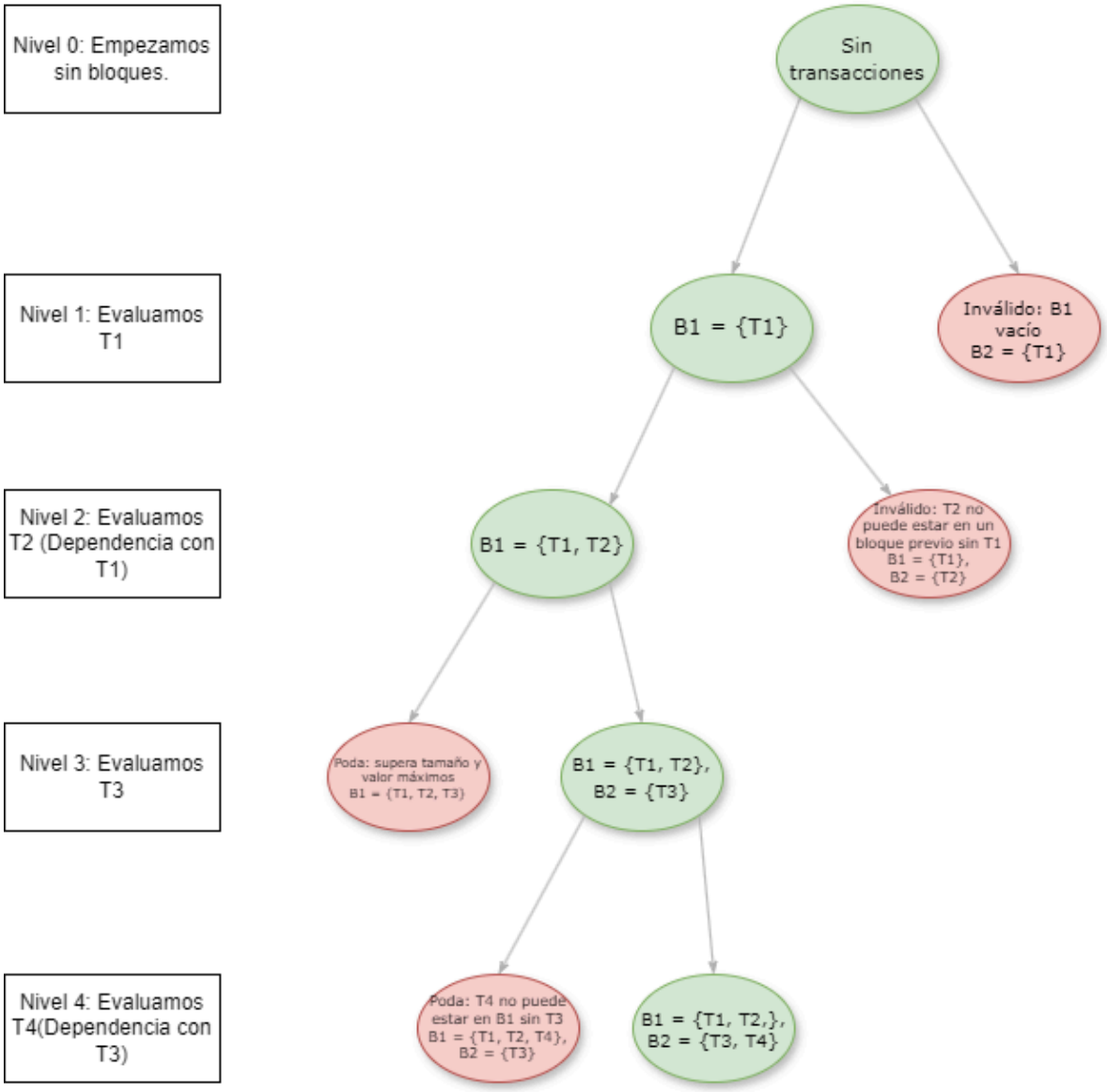
Si una rama de búsqueda no cumple con alguna, se evitará seguir explorando combinaciones partiendo de la configuración actual ya que no serán válidas.
Cuando todas las transacciones fueron asignadas de forma correcta creando una blockchain válida, se guardará la configuración de bloques en la lista de soluciones mientras se siguen buscando más combinaciones válidas.
Esta estrategia puede ser representada mediante el recorrido en profundidad de un árbol donde cada nodo representará una configuración parcial de bloques (cómo fueron distribuidas las transacciones en un momento determinado), y cada artista representará la asignación de una transacción a un bloque existente, o la creación de uno nuevo. En el nodo inicial ninguna transacción fue asignada por lo que la lista de bloques empieza vacía. Y cada nodo hoja representará una blockchain válida o una configuración en la que se alcanzó el máximo de bloques sin asignar todas las transacciones. En el primer caso se almacenará la configuración válida en el listado de soluciones.

Teniendo en cuenta la estrategia planteada si ponemos como ejemplo las siguientes transacciones y restricciones podemos armar el siguiente árbol de evaluación de las combinaciones, indicando en cada nivel la transacción estamos evaluando, cómo se arman las opciones según validaciones del algoritmo y las condiciones de poda.

Con las siguientes transacciones como ejemplo:
T1: Tamaño = 200 KB, Valor = 30 satoshis, Dependencias = Ninguna
T2: Tamaño = 150 KB, Valor = 40 satoshis, Dependencias = T1
T3: Tamaño = 300 KB, Valor = 50 satoshis, Dependencias = Ninguna
T4: Tamaño = 100 KB, Valor = 20 satoshis, Dependencias = T3

Y las siguientes restricciones como ejemplo:
Tamaño máximo del bloque: 1 MB (1024 KB)
Valor máximo del bloque: 100 satoshis
Máximo de transacciones por bloque: 3
Máximo de bloques: 2

Se forma el siguiente árbol de combinaciones:



En color verde podemos identificar los nodos válidos, mientras que en color rojo podemos identificar los caminos inválidos, ya sea por condiciones de poda o validaciones del algoritmo por dependencias entre las transacciones.

Pseudocódigo del Algoritmo de Resolución del Problema

```
Algoritmo construirBlockchain

Entrada:

  List<Transaccion> transacciones

  Entero maxTamañoBloque,

  Entero maxValorBloque,

  EnteromaxTransacciones,

  EnteromaxBloques

Salida: List<Bloque>

    soluciones = []

    backtrack(transacciones, 0, [], soluciones, maxTamañoBloque,
maxValorBloque, maxTransacciones, maxBloques)

    devolver soluciones

fin construirBlockchain


Algoritmo construirBlockchainBacktracking

Entrada:

  List<Transaccion> transacciones,

  Entero index,

  List<Bloque> bloquesActuales,

  List<List<Bloque>> soluciones,

  Entero maxTamañoBloque,

  Entero maxValorBloque,

  Entero maxTransacciones,
```

```
  Entero maxBloques
Salida: -
    si index == transacciones.length():
        # Caso base: todas las transacciones han sido asignadas
        si cumplenPruebaDeTrabajo(bloquesActuales):
          soluciones.agregar(copiar(bloquesActuales))
        fin si
        devolver
    fin si

    # Obtener la transacción actual
    transaccion = transacciones[index]

    mientras !transaccion.isFirmada()
      transaccion.agregarFirma()
    fin mientras

    # Intentar agregar la transacción en cada bloque existente
    por cada bloque en bloquesActuales:
        si cumpleRestricciones(bloque, transaccion,
maxTamañoBloque, maxValorBloque, maxTransacciones):

            List<Transaccion> lista = bloque.getTransacciones()
            lista.agregar(transaccion)
            bloque.setTransacciones(lista)
```

```
            bloque.setTamanioTotal(bloque.getTamanioTotal() +
transaccion.getTamanio())

            bloque.setValorTotal(bloque.getValorTotal() +
transaccion.getValor())


            construirBlockchainBacktracking(transacciones, index
+ 1, bloquesActuales, soluciones, maxTamañoBloque,
maxValorBloque, maxTransacciones, maxBloques)


            lista.pop() # Backtracking

            bloque.setTransacciones(lista)

            bloque.setTamanioTotal(bloque.getTamanioTotal() -
transaccion.getTamanio())

            bloque.setValorTotal(bloque.getValorTotal() -
transaccion.getValor())

        fin si

    fin por


    # Intentar crear un nuevo bloque si no hemos alcanzado el
máximo de bloques

    si bloquesActuales.length() < maxBloques:

        nuevoBloque = new Bloque()

        List<Transaccion> lista = nuevoBloque.getTransacciones()

        lista.agregar(transaccion)

        nuevoBloque.setTransacciones(lista)

        nuevoBloque.setTamanioTotal(transaccion.getTamanio())

        nuevoBloque.setValorTotal(transaccion.getValor())
```

```
        bloquesActuales.agregar(nuevoBloque)


        construirBlockchainBacktracking(transacciones, index + 1,
bloquesActuales, soluciones, maxTamañoBloque, maxValorBloque,
maxTransacciones, maxBloques)


        bloquesActuales.remove(nuevoBloque)  # Backtracking
    fin si
fin construirBlockchainBacktracking


Algoritmo cumpleRestricciones
Entrada:
  Bloque bloque,
  Transaccion transaccion,
  Entero maxTamañoBloque,
  Entero maxValorBloque,
  Entero maxTransacciones
Salida: Booleano
    devolver (
        bloque.getTamanioTotal() + transaccion.getTamanio() <=
maxTamañoBloque Y
        bloque.getValorTotal() + transaccion.getValor() <=
maxValorBloque Y
        bloque.getTransacciones().size() + 1 <= maxTransacciones
Y
        (transaccion.getDependencia() == null O
bloque.contiene(transaccion.getDependencia()))
```

```
    )
fin cumpleRestricciones


Algoritmo cumplenPruebaDeTrabajo
Entrada: List<Bloque> bloques
Salida: Booleano
  por cada bloque en bloques:
    si  bloque.getValorTotal() % 10 != 0
      devolver falso
    fin si
  fin por cada
  devolver verdadero
fin cumplenPruebaDeTrabajo
```

Análisis de Complejidad Temporal

Complejidad Temporal Teórica

Analizando la complejidad temporal del algoritmo **construirBlockchainBacktracking** podemos determinar las siguientes variables presentes:

- El número de transacciones que se asigna a un bloque es **n**.
- El número máximo de bloques es **m**.
- En cada asignación se verifican las validaciones planteadas en cuanto tamaño, valor, dependencias y número máximo de transacciones en el bloque.
- El valor de **a** equivale a **m**, puesto que en cada llamado recursivo la cantidad de subproblemas que se generan depende de la variable “bloquesActuales”, ya que el algoritmo siempre intenta agregar la transacción actual a cada bloque.

- El valor **b** es 1, ya que en cada llamado recursivo el índice aumenta en 1.
- El valor **k** es constante.

Por lo tanto, en cada nivel de recursión, el algoritmo considera entre dos opciones para cada transacción. Se agrega a un bloque existente o crea un nuevo bloque si corresponde con la validación del número máximo de bloques.
Si agrega la transacción a uno de los bloques esto puede generar una ramificación de hasta **m** opciones por transacción en el peor de los casos.
Entonces, esto puede representar una complejidad temporal **exponencial** de aproximadamente **$O(m^n)$** en notación Big O. Ya que para cada transacción se generan **n** opciones.

En el caso de la verificación de las restricciones, se realiza en cada intento de asignación de las transacciones. Todas estas operaciones son constantes y representan una complejidad temporal de **O(1) constante** en notación Big O.

Por otro lado, las operaciones de **backtrack** para deshacer una asignación y evaluar todos las opciones posibles también son **constantes** en cada cada llamado recursivo. Entonces se representa como **O(1)** en notación Big O.

Finalmente, podemos determinar que la complejidad temporal total del algoritmo es **$O(b^n)$** ya que es la complejidad dominante. Tiene sentido ya que el algoritmo explora todas las asignaciones posibles para cada transacción de acuerdo a las restricciones planteadas. Esto determina una cantidad de posibilidades exponencial.

Complejidad Temporal Práctica

Para el cálculo de la complejidad temporal práctica del algoritmo, el mismo se sometió a pruebas con grandes números de transacciones y extendiendo el límite de bloques máximos por solución para poder encontrar los puntos donde la complejidad temporal empieza a crecer de forma exponencial.

Con las mismas restricciones que fueron planteadas en las consignas del TPO respecto a:

- Tamaño del bloque no mayor a 1MB.
- Valor máximo del bloque en 100.
- Cantidad máxima de transacciones por bloque 3.

Para obtener resultados representativos en la prueba, se trabajó sobre el límite de máximos bloques permitidos y una gran cantidad de transacciones, ya que con bajas cantidades de transacciones o límites de bloques los tiempos son casi instantáneos.

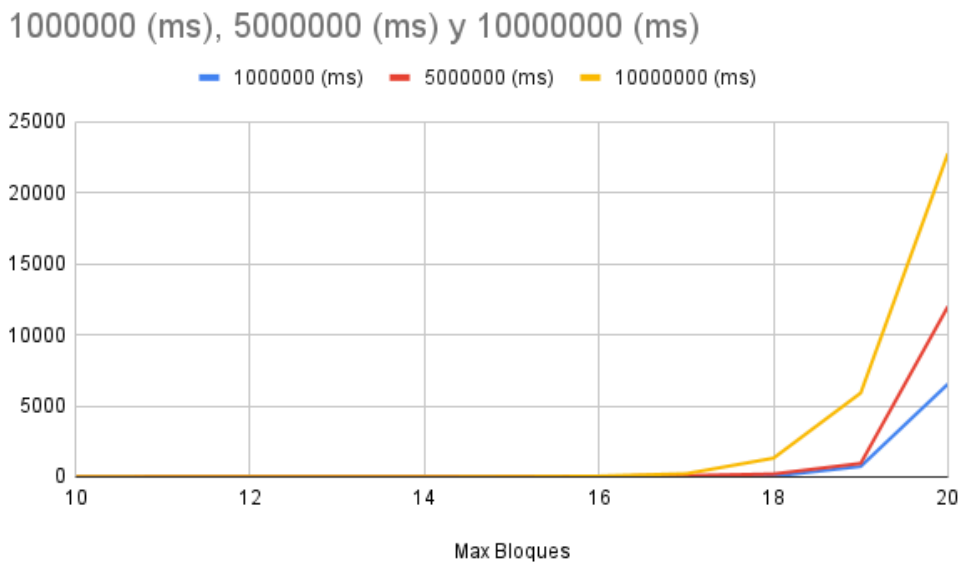
Para esto definimos que las pruebas se realizaron con el siguiente grupo de valores:

- El límite máximo de bloques está en un rango de 10 a 20.
- Se probó el algoritmo con 1.000.000, 5.000.000 y 10.000.000 de transacciones. Todas fueron transacciones simples, sin dependencias entre ellas.

Los resultados obtenidos son en milisegundos el tiempo que tardó el algoritmo en procesar la cantidad de transacciones por cada límite de bloques planteado.

Max Bloques/ N° de Transacciones	1.000.000 (ms)	5.000.000 (ms)	10.000.000 (ms)
10	1	1.2	1.5
11	1.6	1.6	2.5
12	1.6	1.7	6.2
13	4.9	6.7	8.7
14	7.8	8.1	12.5
15	14.1	17.9	21.9
16	21.8	40.6	57.8
17	30.3	90.6	226.5
18	46.9	206.3	1319.3
19	739	950.4	5908.3
20	6540	12000.9	22772.1

Representado en un gráfico lo podemos ver como:



En la prueba y como se pudo observar en el gráfico hay un gran salto a partir del límite de 18 bloques donde los tiempos de procesamiento del algoritmo crecen de forma exponencial. El incremento se ve más notorio al incrementar de 5 millones de transacciones a 10 millones de transacciones. Para este último caso, el tiempo de procesamiento ya se incrementa considerablemente para el límite de 17 bloques.

Conclusiones

En este informe, se aborda el desafío de construir una especie de red de blockchain válida a partir de un conjunto de transacciones bajo condiciones y restricciones específicas. Para resolver este problema, desarrollamos un algoritmo de backtracking que permite explorar exhaustivamente todas las combinaciones posibles de bloques y seleccionar aquellas que cumplen con las condiciones preestablecidas, mientras aplicamos una poda para ir descartando opciones sin sentido optimizando así el tiempo de ejecución.

En el análisis teórico de la complejidad del algoritmo el resultado fue que este tiene una complejidad exponencial $O(m^n)$, demostrado previamente que m es la cantidad de bloques y que por cada uno de estos se podrían llegar a agregar, en el peor de los casos, n transacciones. Dado que nuestro algoritmo implementado en Java fue una réplica del pseudocódigo planteado, en las pruebas de estrés realizadas pudimos concluir que la complejidad práctica de este también era $O(m^n)$.

En conclusión, se puede ver que el algoritmo planteado resuelve efectiva y eficazmente el problema y a su vez pudimos prever correctamente la complejidad temporal del algoritmo que creamos, mediante el uso de la aplicación del teorema maestro para resolver la complejidad temporal de algoritmos recursivos.