FACULTAD DE INGENIERÍA Y CIENCIAS EXACTAS

DEPARTAMENTO DE TECNOLOGÍA INFORMÁTICA

Trabajo Practico Obligatorio - Blockchain

Profesor/es:

Cuadrado Estrebou, Maria Fernanda

Foglino, Alejandro Luis

Grupo N° 5:

Agustin Herrero - 1174588

Jaldin Walter - 1155940

Mauro Losada - 1124705

Lisandro Rodríguez - 1202380

Buenos Aires, 15 de Noviembre de 2024

Tabla de Contenidos

Introducción

Una blockchain es una estructura de datos distribuida que garantiza la integridad de las transacciones mediante reglas criptográficas y consensos entre los nodos.

El objetivo es diseñar una cadena de bloques compleja, donde no solo deben respetarse las reglas básicas de tamaño y validación de hash, sino también deben gestionarse transacciones que dependen entre sí, firmas múltiples, restricciones de prueba de trabajo, y prioridades en las transacciones.

El objetivo es implementar un algoritmo que construya una blockchain válida, considerando todas las combinaciones posibles de bloques. Cada bloque debe cumplir con un conjunto más amplio de reglas, que incluyen validaciones adicionales como la dependencia entre transacciones, transacciones con firmas múltiples, priorización y restricciones de prueba de trabajo (hash).

Descripción del Problema

Estrategia de Resolución

El objetivo de este algoritmo es encontrar todas las combinaciones posibles de transacciones que se puedan distribuir de manera válida en bloques, respetando las restricciones de la blockchain, como el tamaño máximo de bloque, el valor de las transacciones, las dependencias entre ellas, la cantidad máxima de transacciones por bloque, la prueba de trabajo, entre otras.

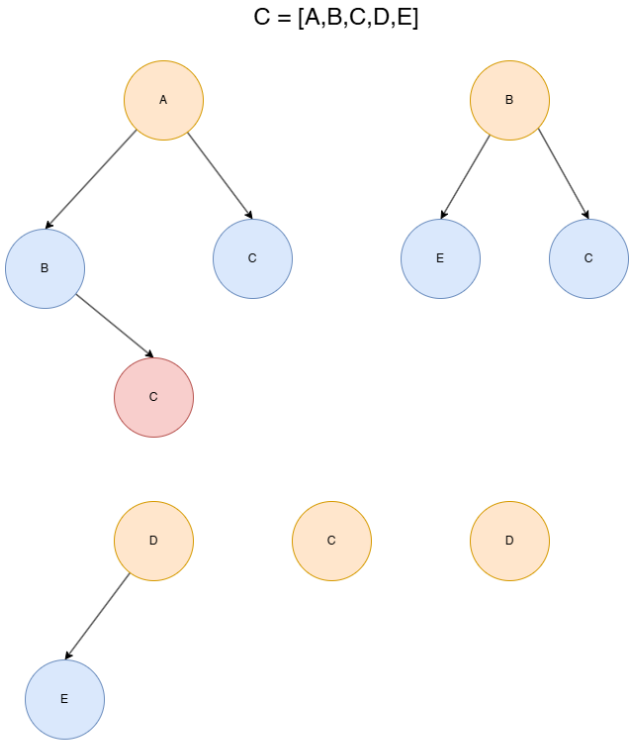
Para lograr esto, se adopta una estrategia de backtracking, que permite explorar todas las posibles combinaciones de transacciones para los bloques de manera eficiente.

- ***Búsqueda de bloques por Niveles:***

Consiste en evaluar cada transacción de manera incremental, desde el primer elemento al último del vector, explorando primero las opciones que determinan un bloque posible.

Comenzamos con una transacción inicial y, para cada transacción posterior, se agrega una nueva transacción al bloque.

Para cada transacción nueva, se verifica si el bloque resultante sigue cumpliendo con las restricciones del problema.



Esta búsqueda resulta en un conjunto de bloque posibles con una cantidad de bloques igual a 2 n siendo n la cantidad de transacciones dadas.

- ***Backtracking de posibles blockchains***

Para cada bloque posible resultante de la búsqueda de bloques por niveles, buscamos todas las soluciones de bloques que pueden combinarse entre sí que respeten las restricciones dadas y que respeten las dependencias entre transacciones.

Para esto, para cada bloque posible, se realizan las combinaciones mediante backtracking podando los bloques que contienen transacciones ya visitadas lo que nos reduce la cantidad de llamados recursivos.

- ***Sin Poda:***

Si el algoritmo no poda, significa que el conjunto de transacciones en todas sus combinaciones posibles son válidas.

- ***Poda:***

En cada paso del backtracking, es necesario verificar que los bloques a añadir no violen ninguna de las restricciones del problema. Esto se realiza mediante la validación de restricciones antes de continuar la búsqueda.

Si el bloque cumple con las restricciones, se sigue explorando el siguiente nivel; si no, se poda esa rama de la búsqueda y se retrocede para intentar agregar otro bloque del conjunto disponible.

- ***Exploración Completa de Soluciones:***

En cada iteración, se exploran todos los bloques que aún no se han considerado y se verifica si al añadirlos al blockchain actual se mantiene el cumplimiento de las restricciones. Si se alcanzan las restricciones máximas (como el número de bloques máximos o que se visitaron todas las transacciones), la blockchain es considerada completa y se agrega a la lista de soluciones.

Pseudocódigo del Algoritmo de Resolución del Problema

```
Algoritmo construirBlockchain

Entrada Vector<objeto> transacciones, int maxTamañoBloque, int
maxValorBloque, int maxTransacciones, int maxBloques, int pruebaDeTrabajo

Salida: Lista<Lista<Bloque>>

soluciones <- []

transacciones_visitadas <- []

transaccionesValidas <- []

bloquesPosibles <- []

combinacionActual <- []
```

```
transaccionesValidas <- validarTransacciones(transacciones, maxValorBloque,
maxTamanioBloque)

bloquesPosibles <- buscarBloquesPosibles(transacionesValidas,
maxTransacciones, maxTamañoBloque, maxValorBloque, pruebaDeTrabajo)

blockchainBacktracking(transaccionesValidas, combinacionActual,
bloquesPosibles, soluciones, transaccionesVisitadas, maxTamañoBloque)

devolver soluciones
```

Algoritmo **verificarDependencias**

```
Entrada Vector<Bloque> combinacionActual, Conjunto<Transaccion>
transaccionesVisitadas

Salida Boolean

para cada bloque en combinacionActual

      para cada transaccion en bloque

            si transaccion.dependencia != null Y
!Existe(transaccionesVisitadas, transaccion.dependencia)

                  devolver Falso

    devolver Verdadero
```

Algoritmo **esBloqueValido**

```
Entrada: Bloque bloque, int maxTamañoBloque, int maxValorBloque, int
maxTransacciones, int pruebaDeTrabajo

Salida: Boolean bloqueValido

bloqueValido <- Falso

si (bloque.valorTotal <= maxTamañoBloque Y bloque.TamañoTotal <=
maxTamañoBloque Y bloque.ValorTotal % pruebaDeTrabajo = 0)

      bloqueValido <- Verdadero

bloqueValido <- Falso
```

```
devolver bloqueValido


Algoritmo buscarBloquesPosibles

Entrada: Vector<Transaccion> transacciones, int maxTransacciones, int
maxTamañoBloque, int maxValorBloque, int pruebaDeTrabajo

Salida: Vector<Bloques> bloquesPosibles

bloquesPosibles <- []

combinacionActual <- []

combinaciones(transacciones, combinacionActual, 0, maxTransacciones,
maxTamañoBloque, maxValorBloque, pruebadeTrabajo)

devolver bloquesPosibles


Algoritmo Combinaciones

Entrada: Vector<Transaccion> transacciones, Vector<Bloque> bloquesPosibles,
Vector<Transaccion> combinacionActual, int inicio, int maxTamanioBloque, int
maxValorBloque, int pruebaDeTrabajo

Salida: -

si !(combinacionActual=null)

      bloque <- new Bloque

      tamanioTotal <- 0

    valorTotal <- 0

    para cada transaccion en combinacionActual

      tamanioTotal <- tamanioTotal + transaccion.Tamaño

        valorTotal <- valorTotal + transaccion.Valor

    bloque.setTamañoTotal(tamañoTotal)

    bloque.setValorTotal(valorTotal)
```

```
    nuevaCombinacionActual <- []
    nuevaCombinacionActual <- combinacionActual
    bloque.setTransacciones(nuevaCombinacionActual)
    si esBloqueValido(bloque, maxTamañoBloque, maxValorBloque,
maxTransacciones, pruebaDeTrabajo)
      Añadir(bloquesPosibles, bloque)
si longitud(combinacionActual) = maxTransacciones
      devolver
para i en longitud(transacciones):
      Añadir(combinacionActual, transacciones[i])
    combinaciones(transacciones, bloquesPosibles, combinacionActual, i + 1,
maxTransacciones, maxTamañoBloque, maxValorBloque, pruebaDeTrabajo)
    Remover(combinacionActual, longitud(combinacionActual)-1)
```

Algoritmo **validarTransacciones**

```
Entrada: Vector<Transaccion> transacciones, int maxValorBloque, int
maxTamañoBloque
transaccionesValidas <- []
para i en longitud(transacciones):
      combinacionActual <- transacciones.get(i)
    si esFirmada(combinacionActual) Y combinacionActual.tamaño <=
maxTamañoBloque Y combinacionActual.valor <= maxValorBloque
      Añadir(transaccionesValidas, combinacionActual)
devolver transaccionesValidas
```

Algoritmo **eliminarBloquesConTransferenciasUtilizadas**

```
Entrada: Vector<Bloque> bloquesPosibles, Set<Transaccion>
transaccionesVisitadas

Salida: Vector<Bloque> bloquesPosiblesActualizados

bloquesPosiblesActualizados <- []

para cada bloque en bloquesPosibles:

      bloqueConTransferenciaVisitada <- Falso

      para cada transaccion en bloque.getTransacciones()

            si !Contiene(transaccionesVisitadas, transaccion)

            bloqueConTransferenciaVisitada <- Verdadero

      si !bloqueConTransferenciaVisitada

      Añadir(bloquesPosiblesActualizados, bloque)

devolver bloquesPosiblesActualizados
```

Algoritmo **BlockchainBacktracking**

```
Entrada: Lista transacciones, Lista combinacionActual, Lista
bloquesPosibles, Lista soluciones, Conjunto transaccionesVisitadas, Entero
maxBloques

Salida: -

    Si combinacionActual !=null

        Si verificarDependencias(combinacionActual, transaccionesVisitadas)=
true

            List<Bloque> listaDeBloques <- combinacionActual

            Añadir(soluciones,listaDeBloques)

            int posUltimoBloque <- Longitud(combinacionActual) - 1

            Bloque ultimoBloque <- combinacionActual.get(posUltimoBloque);

            List<Transaccion> transaccionUltimoBloque <-
ObtenerTransacciones(ultimoBloque)
```

```
            Para transaccion in transaccionUltimoBloque

                Si transaccion not in transaccionesVisitadas

                    Añadir( transaccionesVisitadas,transaccion)

                Fin Si

            Fin Para

            bloquesPosibles <-
eliminarBloquesConTransferenciasUtilizadas(bloquesPosibles,
transaccionesVisitadas)

        Sino

            Return

        Fin Si

    Fin Si

    Si Longitud(transaccionesVisitadas) = Longitud(transacciones) O
Longitud(combinacionActual) >= maxBloques Entonces

        Return

    Fin Si

    Para i en Longitud(bloques Posibles)

        Añadir(Obtener(bloquesPosibles, i), combinacionActual)

        blockchainBacktracking(transacciones, combinacionActual,
bloquesPosibles, soluciones, transaccionesVisitadas, maxBloques)

        int posUltimoBloque <- Longitud(combinacionActual) - 1

        Bloque ultimoBloque <- combinacionActual.get(posUltimoBloque)

        List<Transaccion> transaccionUltimoBloque <-
ultimoBloque.getTransacciones();

        Para cada transaccion in transaccionUltimoBloque

            Si transaccion in transaccionesVisitadas

                Eliminar( transaccionesVisitadas,transaccion)
```

```
            Fin Si
    i++
        Fin Para
        Eliminar( combinacionActual,Longitud(combinacionActual) - 1)
    Fin Para
Fin Algoritmo
```

Análisis de Complejidad Temporal

Llamado recursivo en Combinaciones()

a >= 2:

El valor de a representa el número de transacciones posibles que quedan por agregar en cada nivel de recursión.

En el ciclo for (int i = start; i < transacciones.size(); i++), el rango de índices disponibles asegura que siempre existan

a≥2 opciones al inicio (siempre y cuando haya al menos dos transacciones en la lista inicial transacciones).

b = 1 -> se reduce en 1 caso resta, segun la longitud de la lista transacciones

k = La altura del árbol está determinada por la cantidad máxima de transacciones, que está limitada por maxTransacciones; k=maxTransacciones

La complejidad temporal del algoritmo se expresa como O(n^k), donde n representa el número total de transacciones y k corresponde al número máximo de transacciones permitidas en un bloque

Llamado recursivo en blockchainBacktracking()

*Implica iterar sobre todos los bloques restantes y verificar cada transacción en esos bloques, lo que tiene una complejidad temporal de O(k * b) donde k es el número de bloques y b es el número de transacciones por bloque.*

Dado que las operaciones dentro de una iteración del bucle recursivo son todas constantes, el factor clave que afecta a la complejidad temporal de este algoritmo es el número de bloques k y el número de transacciones b.

Entonces, la complejidad temporal del algoritmo es: O(k·b)

Conclusión complejidad temporal:** el peor de los casos el tiempo de ejecución mayor se da por blockchainBacktracking(), dando el algoritmo final una complejidad temporal de **O(k * b)

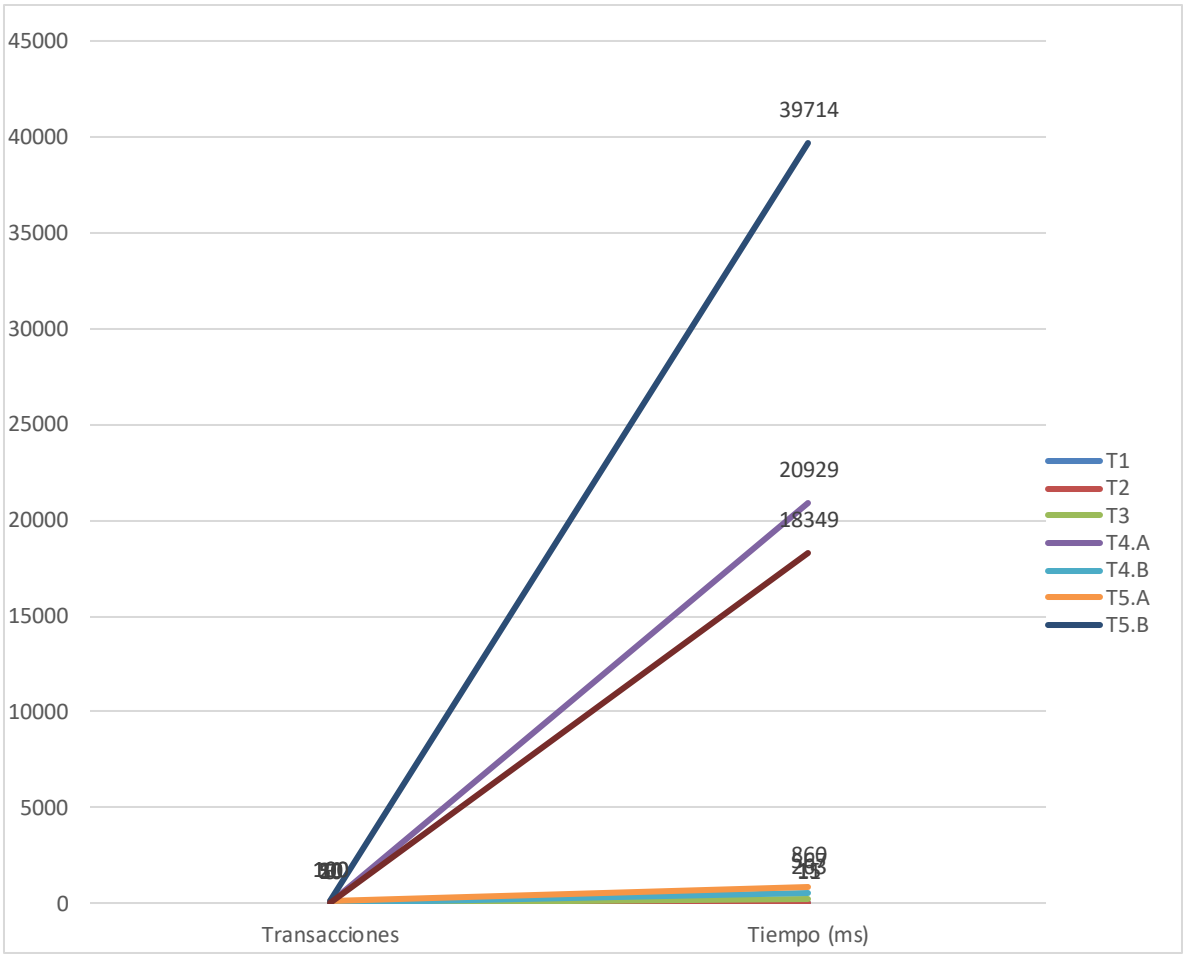
$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \; div \; b}) & \text{si } a > 1 \end{cases}$$

Conclusiones

*El uso de backtracking en esta implementación demostró ser beneficioso para abordar problemas complejos, donde la estrategia y la exploración exhaustiva son esenciales, priorizando la táctica sobre la eficiencia. Al analizar la complejidad teórica del pseudocódigo, se evidenció que el backtracking implica un mayor costo computacional en comparación con otras soluciones más optimizadas, debido a la naturaleza recursiva del método. La complejidad temporal del algoritmo en el peor de los casos se expresa como O(k * b), donde k es el número de bloques y b es el número de transacciones por bloque.*

Durante las pruebas prácticas, se observó que el tiempo de ejecución nunca es constante, ya que depende de las combinaciones posibles.

	T1	T2	T3	T4.A	T4.B	T5.A	T5.B	T6
inicio	3	5	10	20	20	30	30	30
maxTransacciones	2	5	10	30	30	70	70	20
maxTamanioBloque	50	50	50	105	105	105	105	105
maxValorBloque	50	50	50	105	105	105	105	105
pruebaDeTrabajo	0	0	0	0	0	0	0	0
Transacciones	5	10	20	50	50	100	100	50
Tiempo (ms)	11	15	263	20929	567	860	39714	18349



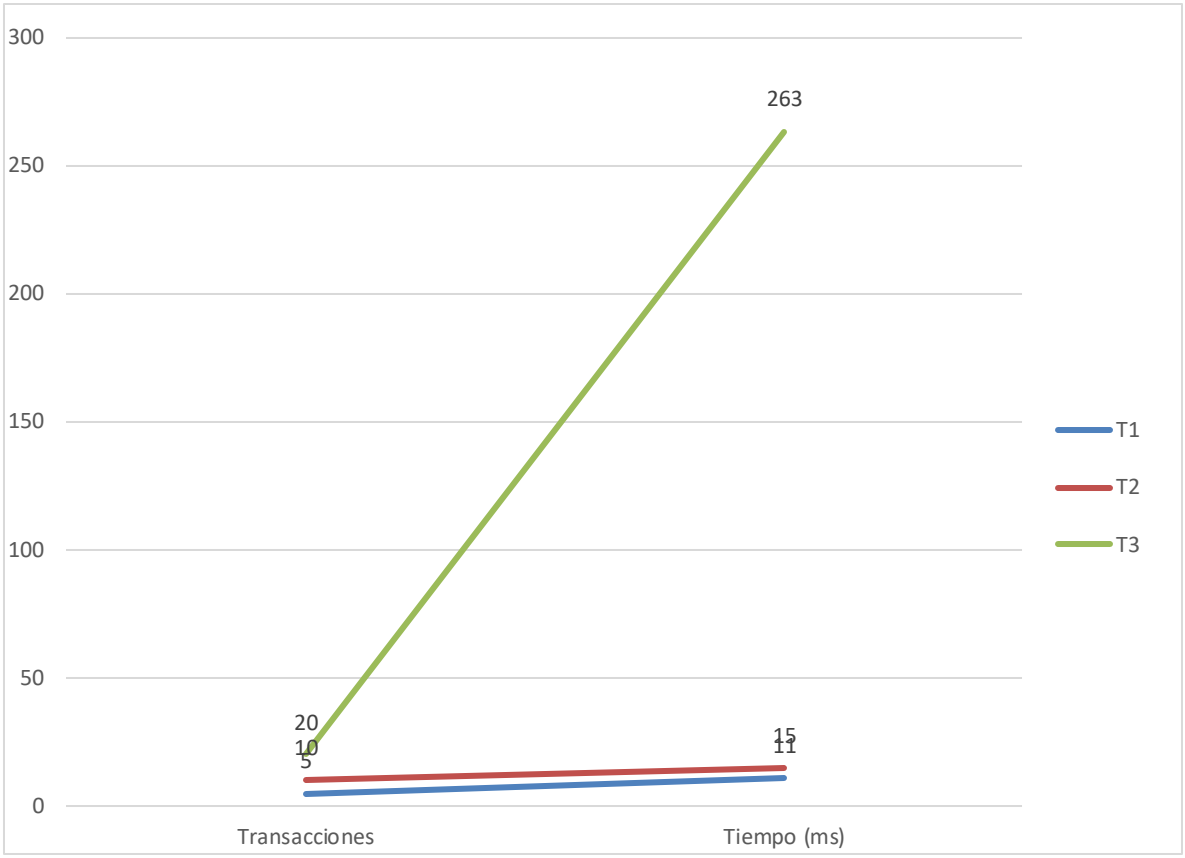
Se evidenció que, mientras más transacciones simples existan en comparación con transacciones que dependan unas de otras, mayor será el tiempo necesario para encontrar soluciones válidas. Por otro lado, si las transacciones tienen un tamaño o valor que supera los límites establecidos, el programa será más eficiente, ya que los bloques inválidos se descartan rápidamente. En cambio, si estas condiciones no están restringidas, el programa deberá evaluar un mayor número de combinaciones, incrementando el tiempo de ejecución. Finalmente, a medida que el número total de transacciones en el blockchain crece, el tiempo de procesamiento aumenta.

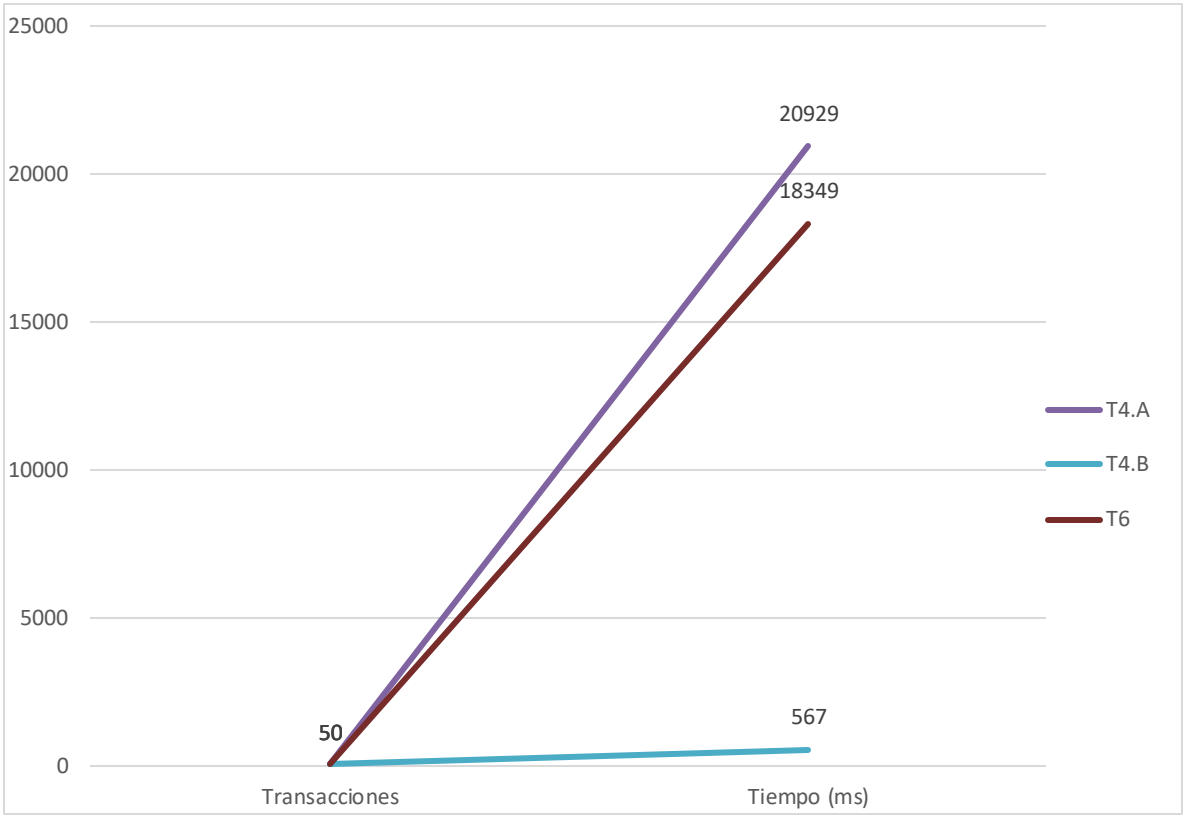
Durante el desarrollo del proyecto, se destacó la importancia de realizar un estudio preciso y actualizaciones constantes. Se debe hacer un estudio detallado para optimizar el código final lo máximo posible. Para esto, tuvimos que observar todas las situaciones posibles en las que la poda es efectiva, como el caso en el que el valor y el tamaño de las

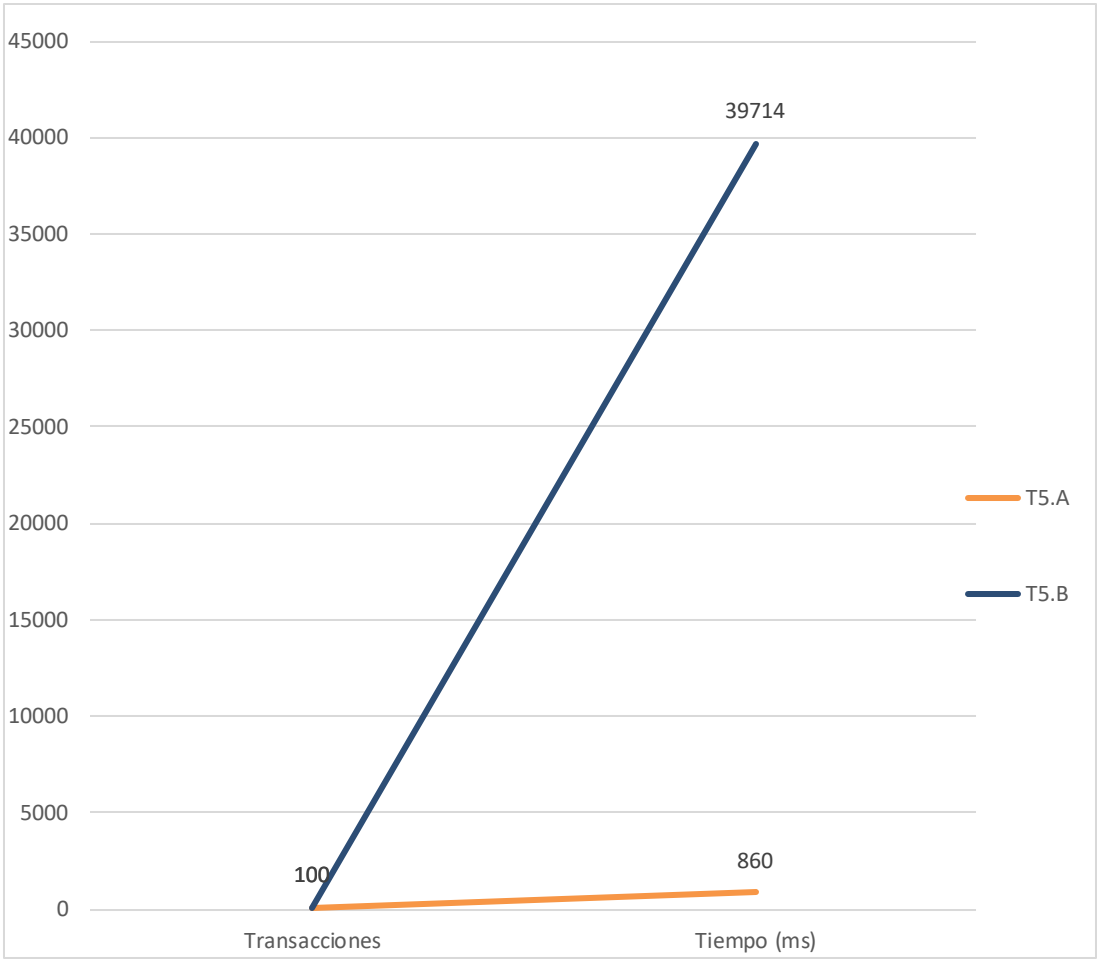
transacciones no excedan lo permitido por bloque, o que la suma total de los valores de un bloque sea divisible por la prueba de trabajo dada. Además, fue necesario realizar actualizaciones al pseudocódigo. Al escribirlo, pudimos observar que cometimos errores y nos faltaron detalles en la descripción de ciertos métodos.

En conclusión, aunque el backtracking enfrenta dificultades en cuanto a eficiencia, resulta útil en situaciones que involucran combinaciones o participaciones (en nuestro caso, transacciones visitadas), ya que cuando se llega a una situación en la que la combinación actual no cumple con las restricciones, en lugar de seguir intentando caminos erróneos, el algoritmo retrocede para intentar una opción distinta.

Gráficos Aumentados Complejidad Practica




25000
20000
15000
10000
5000
0
20929
18349
50
567
Transacciones
Tiempo (ms)
T4.A
T4.B
T6


45000
40000
35000
30000
25000
20000
15000
10000
5000
0
39714
100
860
Transacciones
Tiempo (ms)
T5.A
T5.B

Bibliografía

- Fundamentals of algorithmics - BRASSARD, Gilles - Prentice Hall -1996.