Trabajo Práctico Integrador Programación III

Profesor/es:

Cuadrado Estrevou, María Fernanda

Foglino, Alejandro Luis

Grupo:

Benito, Franco – LU: 1088411

Velázquez, Joaquín– LU: 1186723

Saiburo, Juan Manuel – LU: 1021142

Cataldi, Giuliana – LU: 1149419

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

Entradas:

Un conjunto de transacciones, donde cada una tiene:

- Un tamaño en KB.
- Un valor en criptomoneda (por ejemplo, en satoshis).
- Un conjunto de dependencias: algunas transacciones solo pueden ser procesadas si ciertas otras ya han sido incluidas en bloques anteriores.
- Multifirma: algunas transacciones requieren varias firmas antes de ser añadidas al bloque.

Un conjunto de reglas blockchain:

- El tamaño máximo de un bloque es de 1 MB.
- Cada bloque debe cumplir con una prueba de trabajo (por ejemplo, un hash que comienza con un número determinado de ceros).
- La blockchain puede tener un número dinámico de bloques basado en el tamaño total de las transacciones.

Salidas:

Todas las blockchains válidas que cumplan con las restricciones, incluyendo las soluciones donde las transacciones son distribuidas en bloques de acuerdo con las reglas establecidas.

Validación:

1. Valor máximo por bloque: La suma de los valores de las transacciones en un bloque no debe superar 100 satoshis.
2. Máximo de transacciones por bloque: Un bloque puede contener como máximo 3 transacciones.
3. Prueba de trabajo: La suma de los valores de las transacciones en un bloque debe ser divisible por 10.
4. Tamaño máximo de bloque: El tamaño de cada bloque sigue siendo de 1 MB.

Estrategia de Resolución

La estrategia seleccionada es la de un algoritmo clásico de Backtracking

1.Inicialización y preparación

- Entrada: Recibimos una lista de transacciones, donde cada transacción tiene atributos como tamaño en KB, valor en satoshis, dependencias y firmas requeridas.
- Preparación: Creamos un conjunto para almacenar todas las combinaciones válidas de bloques. Inicializamos una lista auxiliar para almacenar las transacciones de un bloque en construcción. También definimos las reglas de validación (como el tamaño máximo del bloque, valor máximo en satoshis, etc.) y las dependencias entre transacciones.

2.Iteración

- Iteramos sobre todas las transacciones disponibles para agregar al bloque que estamos construyendo. Durante cada iteracion se lleva a cabo:

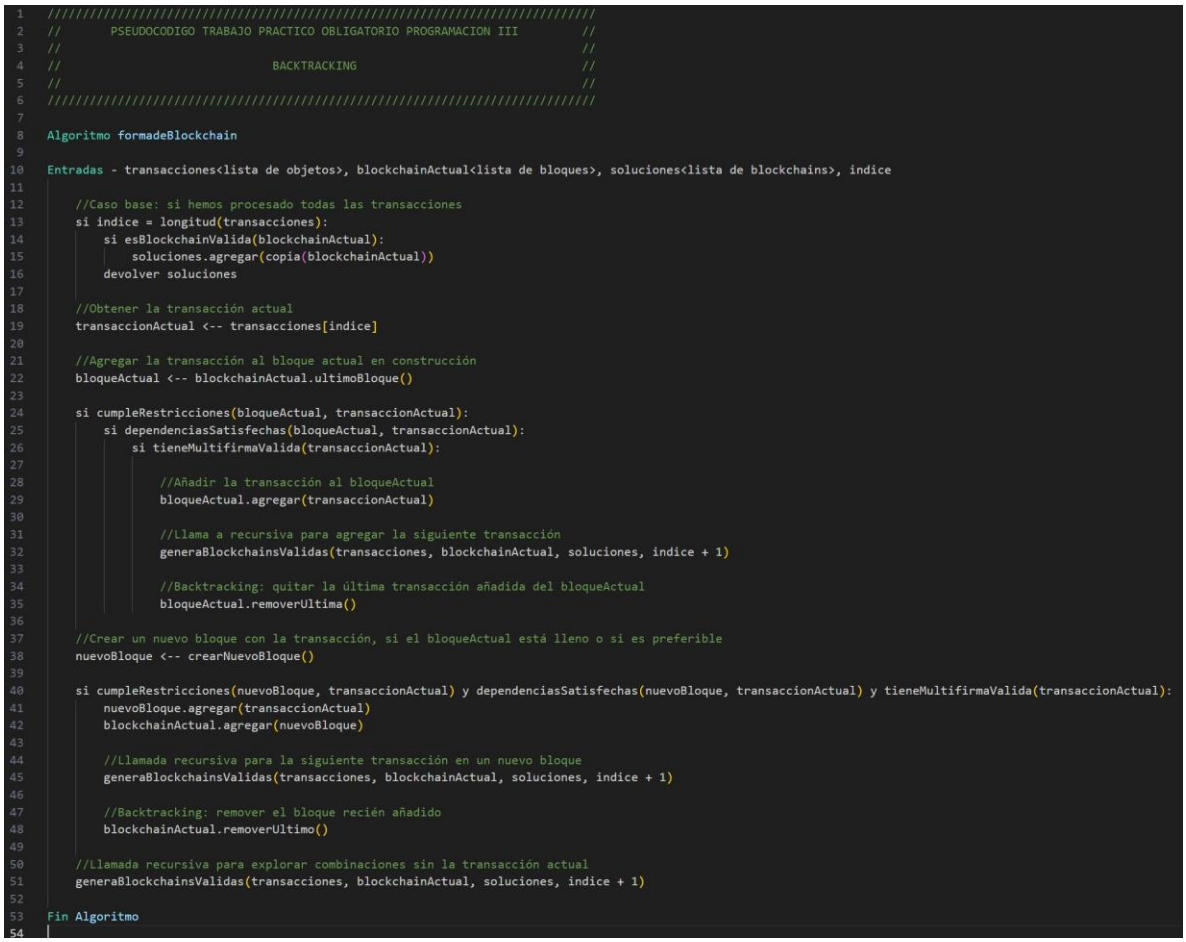
    i. Verificacion de restricciones: Nos aseguramos de que la transaccion que estamos analizando cumpla con las condiciones y requerimientos establecidos.

ii. Control de dependencias: Verificamos si la transaccion cumple con sus dependencias. Por ejemplo, si la transaccion n+1 requiere de la transaccion n, asegurarnos que la misma se encuentre en un bloque anterior.

3.Recursividad y casos de poda

- Caso base (condición de poda): Cuando alcanzamos el final de la lista de transacciones o no hay más combinaciones posibles, añadimos la combinación actual de bloques a nuestra lista de soluciones válidas y erminamos la rama de recursión.
- Caso recursivo: Si una transacción cumple las restricciones, la agregamos al bloque actual y llamamos recursivamente a la funcion para intentar agregar la siguiente transacción. Si al agregar una transacción nueva no se cumplen los requerimientos, retrocedemos en el arbol de ejecución, removemos la última transacción y probamos con otra combinación.

Pseudocódigo del Algoritmo de Resolución del Problema



```
////////////////////////////////////////////////////////////////////////////
//       PSEUDOCODIGO TRABAJO PRACTICO OBLIGATORIO PROGRAMACION III       //
//                                                                        //
//                            BACKTRACKING                                //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

Algoritmo formadeBlockchain

Entradas - transacciones<lista de objetos>, blockchainActual<lista de bloques>, soluciones<lista de blockchains>, indice

    //Caso base: si hemos procesado todas las transacciones
    si indice = longitud(transacciones):
        si esBlockchainValida(blockchainActual):
            soluciones.agregar(copia(blockchainActual))
        devolver soluciones

    //Obtener la transacción actual
    transaccionActual <-- transacciones[indice]

    //Agregar la transacción al bloque actual en construcción
    bloqueActual <-- blockchainActual.ultimoBloque()

    si cumpleRestricciones(bloqueActual, transaccionActual):
        si dependenciasSatisfechas(bloqueActual, transaccionActual):
            si tieneMultifirmaValida(transaccionActual):

                //Añadir la transacción al bloqueActual
                bloqueActual.agregar(transaccionActual)

                //Llama a recursiva para agregar la siguiente transacción
                generaBlockchainsValidas(transacciones, blockchainActual, soluciones, indice + 1)

                //Backtracking: quitar la última transacción añadida del bloqueActual
                bloqueActual.removerUltima()

    //Crear un nuevo bloque con la transacción, si el bloqueActual está lleno o si es preferible
    nuevoBloque <-- crearNuevoBloque()

    si cumpleRestricciones(nuevoBloque, transaccionActual) y dependenciasSatisfechas(nuevoBloque, transaccionActual) y tieneMultifirmaValida(transaccionActual):
        nuevoBloque.agregar(transaccionActual)
        blockchainActual.agregar(nuevoBloque)

        //Llamada recursiva para la siguiente transacción en un nuevo bloque
        generaBlockchainsValidas(transacciones, blockchainActual, soluciones, indice + 1)

        //Backtracking: remover el bloque recién añadido
        blockchainActual.removerUltimo()

    //Llamada recursiva para explorar combinaciones sin la transacción actual
    generaBlockchainsValidas(transacciones, blockchainActual, soluciones, indice + 1)

Fin Algoritmo
```

El pseudocodigo correspondiente a la imagen previa fue alterado al momento de realizar la aplicación final ya que no se tomaron en cuenta ciertas condiciones de poda planteadas en la estrategia inical. Tras un retrabajo, podemos identificar las siguientes condiciones de poda que se aplican al código final:

Límite de tamaño del bloque (Condición 1)
Poda las ramas donde el tamaño del bloque excede 1 MB.
Terminación temprana cuando agregar una transacción superaría el límite de tamaño.
Fórmula: bloque.getTamanoTotal() > maxTamanoBloque * KB_TO_MB

Límite de valor (Condición 2)
Poda las ramas donde el valor del bloque excede 100 satoshis.
Terminación temprana cuando el valor total superaría el límite.
Fórmula: bloque.getValorTotal() > maxValorBloque

Cantidad de transacciones (Condición 3)
Poda las ramas donde el número de transacciones excede 3 por bloque.
Fórmula: bloque.getTransacciones().size() > maxTransacciones

Prueba de trabajo (Condición 4)
Poda los bloques donde el valor total no es divisible por 10.
Fórmula: bloque.getValorTotal() % 10 != 0

Prevención de duplicados (Condición 5)
Poda las ramas que incluirían transacciones ya procesadas.
Fórmula: transaccionesIncluidas.contains(t.getId())

Verificación de dependencias (Condición 6)
Poda las ramas donde no se cumplen las dependencias de las transacciones.
Fórmula: !dependenciasSatisfechas(t, transaccionesIncluidas)

Capacidad del bloque (Condición 7)
Poda las transacciones que no cabrían en el bloque actual según todas las restricciones.
Comprobación combinada en el método cabeEnBloque.

Solución encontrada (Condición 8)
Termina la rama cuando todas las transacciones se incluyen con éxito.
Fórmula: transaccionesIncluidas.size() == transacciones.size()

Máximo de bloques (Condición 9)
Poda las ramas que excederían el número máximo permitido de bloques.
Fórmula: blockchainActual.size() >= maxBloques

Existen multiples razones para la existencia de estas nuevas condiciones de poda.

Eficiencia en el uso de recursos:
Al podar ramas innecesarias o inválidas, se evita gastar tiempo y recursos en cálculos inútiles. Por ejemplo, limitar el tamaño del bloque (Condición 1) asegura que el algoritmo no explore opciones que inevitablemente serán rechazadas por superar el límite permitido.

Cumplimiento de restricciones lógicas y operativas:
Condiciones como la Prueba de Trabajo (Condición 4) y la Prevención de Duplicados (Condición 5) garantizan que los bloques generados no solo sean válidos según las reglas del sistema, sino que también respeten principios como la unicidad de las transacciones y la divisibilidad requerida.

Mantenimiento de la consistencia del sistema:
Verificar dependencias (Condición 6) asegura que las transacciones que dependen de otras sean procesadas en el orden correcto, evitando inconsistencias o errores en los datos.

Optimización del objetivo final:
La detección temprana de soluciones válidas (Condición 8) permite finalizar el proceso cuando se alcanza una meta, evitando recorrer ramas innecesarias.

Control del crecimiento de datos:
El límite en el número máximo de bloques (Condición 9) y la capacidad del bloque (Condición 7) mantienen el sistema manejable, evitando un crecimiento descontrolado que podría afectar su rendimiento.

Análisis de Complejidad Temporal

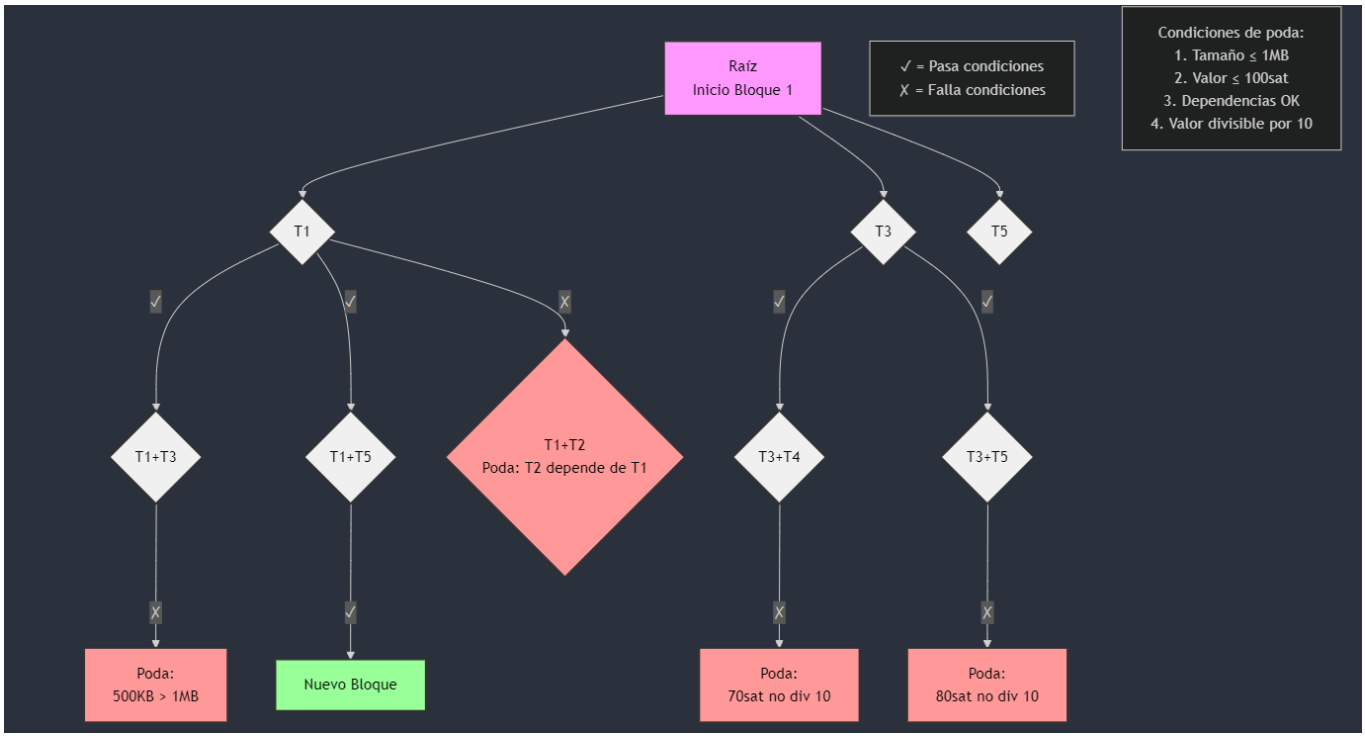
Com plejidad Temporal Teórica

Al realizar un análisis de la complejidad temporal teórica del pseudocodigo , podemos observar que el mismo posee una complejidad temporal de O(n^2) en su peor caso tomando a n como como el numero de transacciones.

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

Esto se debe a que nos encontramos con un caso de sustracción, ya que en cada llamada recursiva estaremos eliminando un elemento de la lista original. Vale decir, que dentro de nuestros parametros, el valor de “b” es igual a 1. Así mismo el valor de a corresponde tambien a 1 por su única llamada recursiva durante la ejecución del algoritmo en el peor de los casos. Finalmente, el valor del parámetro “k”, correspondiente a la complejidad temporal de las operaciones ejecutadas fuera del algoritmo, será igual a 1, ya que ésta corresponde a un ciclo for, el cual refleja una complejidad temporal de O(n), que se traduce a un 1 en el parámetro indicado.

A continuación, graficamos el árbol de ejecución del algoritmo aplicado junto a su descripción.



Estructura del árbol:

- Nivel 1: Elección inicial (T1, T3, T5)
- Nivel 2: Segunda transacción para el bloque
- Las ramas se marcan con ✓ o ✗ para indicar si pasan las condiciones

Ejemplos de poda:

- T1+T3: Se poda por exceder tamaño máximo (500KB > 1MB)
- T1+T2: Se poda inmediatamente por dependencia no satisfecha
- T3+T4 y T3+T5: Se podan por valor no divisible por 10

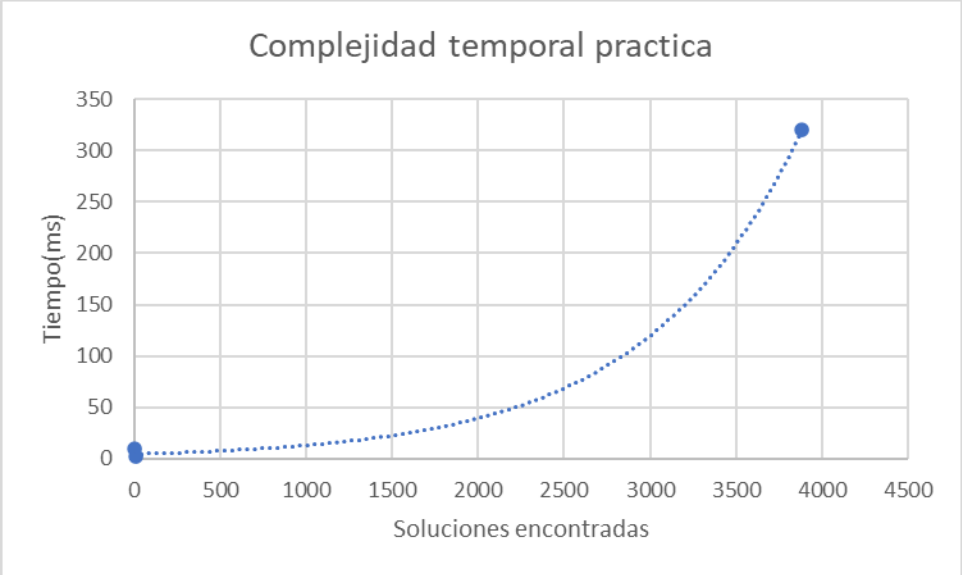
Casos válidos:

- T1+T5 progresa a un nuevo bloque al cumplir todas las condiciones

El árbol muestra cómo cada nivel de decisión está sujeto a todas las condiciones de poda, y cómo una rama se termina tan pronto como falla cualquier condición.

Complejidad Temporal Práctica

Se realizaron una serie de pruebas aleatorizando las condiciones de transacciones y bloque, valores, dependencias, entre otros. El tiempo registrado y plasmado en el gráfico es el promedio de las múltiples pruebas realizadas. Se pude discernir un incremento de tiempo relativo al tamaño de las soluciones encontradas. No se puede tomar con completa validez, ya que otras operaciones de distintos programas de fondo que bloquean recursos del sistema al correr las pruebas afectan los resultados en cuestión.



No pudimos disponer de un sistema asilado para poder realizar pruebas sin dichas intervenciones.

Cabe destacar que se utilizaron valores que permitan visualizar la diferencia de costo temporal para un caso fallido como para sucesos con pocas y muchas soluciones encontradas, satisfaciendo el requerimiento de distintos tamaños de prueba. En las siguienes imágenes, podemos apreciar los distintos casos descriptos junto a su tiempo de ejecución, el cual se ve influenciado por distintos factores que se explayaran en la conclusion.

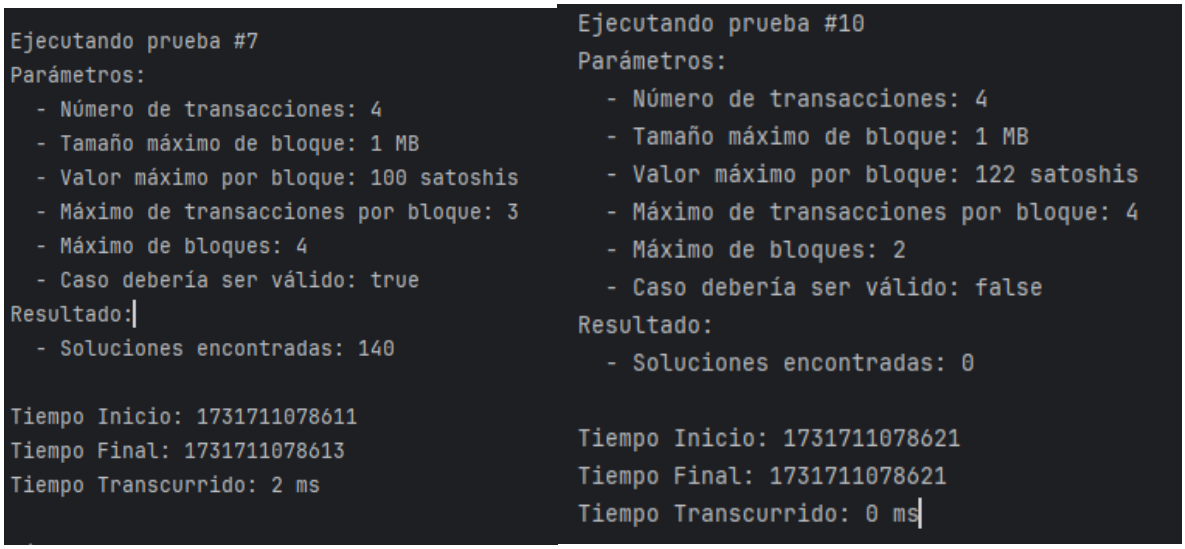
Captura de resultados prácticos realizados

```
Ejecutando prueba #1
Parámetros:
  - Número de transacciones: 8
  - Tamaño máximo de bloque: 1 MB
  - Valor máximo por bloque: 89 satoshis
  - Máximo de transacciones por bloque: 3
  - Máximo de bloques: 4
  - Caso debería ser válido: false
Resultado:
  - Soluciones encontradas: 0

Tiempo Inicio: 1731711078220
Tiempo Final: 1731711078229
Tiempo Transcurrido: 9 ms

Ejecutando prueba #2
Parámetros:
  - Número de transacciones: 9
  - Tamaño máximo de bloque: 1 MB
  - Valor máximo por bloque: 100 satoshis
  - Máximo de transacciones por bloque: 3
  - Máximo de bloques: 4
  - Caso debería ser válido: true
Resultado:
  - Soluciones encontradas: 3888

Tiempo Inicio: 1731711078241
Tiempo Final: 1731711078561
Tiempo Transcurrido: 320 ms
```



Conclusión

Habiendo trasladado la estrategia de un marco teórico a uno práctico en su implementación en Java, pudimos obtener resultados no representativos con lo planificado teóricamente. Esto se debe a una variedad de factores. Entre ellos :

- Recursos compartidos bloueados por la computadora.
- Memoria limitada.
- Reutilización de buffers.
- JIT beneficiado de ejecuciones subsecuentes por reutilización del código.

Aunque estos resultados no representan el costo temporal estipulado en el estudio teórico, sí podemos verificar el correcto funcionamiento del algoritmo implementado en el programa para poder hallar el resultado esperado en base a las entradas que le fuimos otorgando.

Bibliografía

- UVA 6 - Algoritmos de backtracking.
- UVA 7 - Algoritmos de backtracking - Backtracking de Juegos.