UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE CIENCIA

Departamento de Matemática y Computación

Mejoramiento en la Asignación de Tareas de Algoritmos Distribuidos

Claudio Saji Santander

Profesor guía: Dr. Rubén Carvajal Schiaffino

Trabajo de Titulación presentado en conformidad a los requisitos para obtener el título de Analista en Computación Científica

Santiago - Chile

2022

RESUMEN

El problema de la asignación de tareas en un sistema distribuido consiste en distribuir de manera eficiente las tareas que el sistema debe realizar, entre sus distintos procesos, de forma tal que cada uno tenga una carga de tareas con un costo computacional similar. Cada tarea debe ser representada como un número entero mayor a cero, según su costo computacional. Este costo computacional se puede calcular con la subtarea que tiene la mayor complejidad de tiempo.

En el presente trabajo se proponen tres algoritmos para resolver el problema de la asignación de tareas en sistemas distribuidos: un algoritmo implementa la técnica de *backtracking*, y los otros dos son soluciones nuevas, llamados “Zigzag” y “ColFirst”. El algoritmo de *backtracking* tiene una complejidad de tiempo factorial en el peor de los casos, mientras que Zigzag y ColFirst tienen una complejidad de tiempo cuadrática. Se observó además, en el desarrollo de los experimentos, que los tres algoritmos propuestos tuvieron tiempos de ejecución bastante menores de los que deberían tener teniendo en cuenta sus complejidades de tiempo.

El algoritmo de Zigzag fue aplicado para mejorar la asignación de tareas en el algoritmo distribuido para la prueba de normalidad exhaustiva (Carvajal y Moreno, 2022), lo que ayudó a reducir de forma considerable la diferencia entre el proceso que más se demoraba y el proceso que menos se demoraba. Específicamente, en un caso genérico, esta diferencia se redujo de 3984 segundos, a sólo 134 segundos.

Palabras clave: algoritmos distribuidos, optimización.

ABSTRACT

The problem of assigning tasks in a distributed system consists in distributing the tasks that the system must carry out in an equitably way between the different processes of the system, in such a way that all the processes have a load of tasks with a similar computational cost. Each task must be represented as an integer greater than zero, based on its computational cost. This computational cost can be obtained using the subtask that has the highest time complexity.

The present work proposes three algorithms to solve the task assignment problem in distributed systems: one algorithm implements a backtracking technique, and the other two are new solutions, called “Zigzag” and “ ColFirst”. The backtracking algorithm has a factorial time complexity, while Zigzag and ColFirst have quadratic time complexity. However, in the development of the experiments, the three proposed algorithms had execution times that were much lower than they should have been, considering their time complexities.

The Zigzag algorithm was applied to improve the assignment of tasks in the distributed algorithm for the exhaustive normality test (Carvajal y Moreno, 2022), which helped to significantly reduce the difference between the process that took the maximum amount of time and the process that took the least amount of time. Specifically, in a generic case, this difference went down from 3984 seconds to only 134 seconds.

Keywords: distributed algorithms, optimization.

Agradecimientos

Agradezco a mis padres por su apoyo y por confiar en mi.

Agradezco a mi profesor guía Rubén Carvajal por su apoyo y consejos, y a mi profesor supervisor Francisco Moreno por su ayuda en la definición del problema.

Por último agradezco a los amigos que tuve en la Universidad por su compañía.

Índice general

Índice de tablas

Índice de figuras

Introducción

Los sistemas distribuidos son una colección de computadores interconectados que se comunican y coordinan sus acciones para lograr un objetivo común. En un sistema distribuido, las tareas generalmente se dividen entre varios computadores para mejorar el rendimiento y la escalabilidad. Sin embargo, la asignación de tareas a los distintos nodos puede ser un problema desafiante, especialmente en sistemas a gran escala, en donde, si no se asignan las tareas de forma correcta, habrá nodos que se demorarán mucho más tiempo en terminar sus tareas que otros nodos; lo que conlleva a que los nodos que ya terminaron sus tareas deban esperar a que los demás nodos terminen las suyas para que finalice todo el sistema distribuido. En cambio, con una correcta asignación de tareas a los distintos nodos, se puede conseguir que todo el sistema distribuido se demore mucho menos en terminar un determinado problema.

El problema de asignación de tareas puede entrar en la categoría de un problema de investigación de operaciones debido a que se busca minimizar las diferencias de tiempo entre los distintos procesos. Específicamente, el problema de este trabajo puede categorizarse como un problema de programación entera, ya que se utilizan números enteros para representar a las tareas que deben realizar los nodos. Sin embargo, debido a que este trabajo de título es uno de ciencia de la computación, no se resolverá el problema con técnicas de programación entera, sino que se resolverá con técnicas de ciencia de la computación. De todas formas se investigará sobre la factibilidad de resolver el problema con técnicas de programación entera, especialmente para conocer la complejidad de tiempo que tendría un algoritmo que implementa estas técnicas.

A simple vista, el problema puede ser resuelto implementando un algoritmo del tipo “backtracking” para explorar todas las permutaciones posibles de los números hasta llegar a una solución. Sin embargo, teniendo en cuenta la complejidad de tiempo que poseen los algoritmos de este tipo, se diseñará e implementará un algoritmo más eficiente.

El problema surge de la necesidad de mejorar la asignación de tareas en el algoritmo distribuido para la prueba de normalidad exhaustiva, diseñado por Carvajal y Moreno (2022). No obstante, la mejora en la asignación puede aplicarse a cualquier sistema distribuido que posea características similares al algoritmo distribuido para la prueba de normalidad exhaustiva.

Estructura del escrito

En el capítulo 1 se explica la programación entera y se da un ejemplo de un problema. Además se explican los algoritmos del tipo “backtracking”, y se presenta el típico problema de las n reinas. Luego, se explica la distribución normal multivariante y los tests de normalidad multivariante, para entender mejor el algoritmo distribuido para la prueba de normalidad exhaustiva, en el cual se aplica la mejora en la asignación de tareas.
En el capítulo 2 se detalla el problema de la asignación de tareas desde un punto de vista matemático. Después, se describen los algoritmos propuestos para resolver el problema.
En el capítulo 3 se describe la asignación de tareas original que tenía el algoritmo distribuido para la prueba de normalidad exhaustiva, y la asignación de tareas mejorada que se consiguió con los algoritmos propuestos. Luego, se presentan los experimentos realizados para los algoritmos propuestos y para el algoritmo distribuido para la prueba de normalidad exhaustiva.
Finalmente, se dan las conclusiones del trabajo realizado, se presenta la bibliografía del escrito, y el apéndice A presenta el código del programa escrito en C, además del código del $script$ utilizado para los experimentos.

Objetivo general

Resolver el problema de la asignación de tareas en sistemas distribuidos.

Objetivos específicos

- Investigar sobre la factibilidad de resolver el problema con técnicas de programación entera.
- Implementar un algoritmo del tipo *backtracking* para resolver el problema.
- Diseñar e implementar un algoritmo más eficiente que el de *backtracking*.
- Aplicar el algoritmo propuesto para mejorar la asignación de tareas en el algoritmo distribuido para la prueba de normalidad exhaustiva.
- Diseñar y ejecutar las pruebas y experimentos.

Capítulo 1

Marco teórico

1.1. Programación entera

Un problema de programación lineal (LP) es una clase del problema de programación matemática, un problema de optimización con restricciones, en el que se busca encontrar un conjunto de valores para variables continuas $(x_1, x_2, ..., x_n)$ que maximiza o minimiza una función objetivo lineal z, mientras se satisface un conjunto de restricciones lineales (un sistema de ecuaciones y/o desigualdades lineales simultáneas) (Chen, Batson, y Dang, 2010). Matemáticamente, un LP se expresa de la siguiente manera:

$$\text{Maximizar } z = \sum_{j=1}^{n} c_j x_j$$

Sujeta a:

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \quad (i = 1, 2, ..., m),$$
$$x_j \geq 0 \quad (j = 1, 2, ..., n),$$
$$x_j \text{ entero} \quad (\text{para algunos o todos } j = 1, 2, ..., n \text{ })$$

Un problema de programación (lineal) entera (IP) es un problema de programación lineal en el que al menos una de las variables está restringida a valores enteros. El término “programación” en este contexto significa actividades de planificación que consumen recursos y/o cumplen requisitos, como se expresa en las m restricciones. Los recursos pueden incluir materias primas, máquinas, equipos, instalaciones, mano de obra, dinero, administración, tecnología de la información, etc. En el mundo real, estos recursos suelen ser limitados y deben compartirse con varias actividades competidoras. Los requisitos pueden imponerse implícita o explícitamente. El objetivo de la LP/IP es asignar los recursos compartidos y la responsabilidad de cumplir con los requisitos a todas las actividades en competencia de manera óptima.

Teóricamente, cualquier problema de IP puro con límites finitos en variables enteras puede resolverse enumerando todas las combinaciones posibles de valores enteros y determinando una combinación (solución) que satisfaga todas las restricciones y produzca el valor objetivo máximo (o mínimo), de ahí el nombre de "enumeración completa". Desafortunadamente, el número de todas las combinaciones posibles es excesivamente grande para ser evaluado incluso para un problema pequeño. Un problema de n variables enteras con m valores cada una tiene un total de m^n combinaciones posibles (soluciones factibles y no factibles). Por lo tanto, la enumeración completa es teóricamente simple pero prácticamente intratable.

Como mejor alternativa, la "enumeración implícita" aplica un esquema de enumeración inteligente que puede cubrir todas las soluciones posibles evaluando explícitamente solo una pequeña cantidad de soluciones mientras ignora (o enumera implícitamente) una gran cantidad de soluciones inferiores. Una de esas estrategias se llama "dividir y conquistar". Básicamente, esta estrategia divide el problema dado en una serie de subproblemas más fáciles de resolver que se generan y resuelven (o superan) sistemáticamente. Las soluciones de estos subproblemas generados se juntan para resolver el problema original.

Una técnica que aplica "dividir y conquistar" es la llamada "Branch-and-bound", la cual puede resolver problemas de IP. Esta técnica realiza un proceso de ramificación para dividir y un proceso de delimitación para conquistar. A lo largo del algoritmo, se generan y resuelven sistemáticamente una serie de subproblemas de LP. Luego, los límites superior e inferior se ajustan progresivamente al valor objetivo del problema de IP original.

Una forma típica de representar un proceso de este tipo es a través de un árbol de ramas y límites, que es un árbol de enumeración especializado para realizar un seguimiento de cómo se generan y resuelven los subproblemas de LP.

1.2. Backtracking

El Backtracking es una técnica de diseño de algoritmos cuya idea principal es construir soluciones por un componente a la vez y evaluar candidatos parcialmente construidos de la siguiente manera. Si una solución parcialmente construida puede desarrollarse más sin violar las restricciones del problema, se hace tomando la primera opción legítima restante para el siguiente componente. Si no hay una opción legítima para el siguiente componente, no es necesario considerar alternativas para ningún componente restante. En este caso, el algoritmo vuelve atrás para reemplazar el último componente de la solución parcialmente construida con su siguiente opción (Levitin, 2012). Este tipo de procesamiento se puede implementar construyendo un árbol de elecciones, llamado árbol de espacio de estado. Su raíz representa un estado inicial antes de que comience la búsqueda de una solución. Los nodos del primer nivel en el árbol representan las elecciones rea-

lizadas para el primer componente de una solución, los nodos del segundo nivel representan las elecciones para el segundo componente, y así sucesivamente. Las hojas representan callejones sin salida o soluciones completas encontradas por el algoritmo. El árbol de espacio de estado se construye mediante una búsqueda en profundidad. Si el nodo actual puede conducir a una solución, su hijo se genera agregando la primera opción legítima restante para el siguiente componente de una solución, y el procesamiento pasa a este hijo. Si se detecta que el nodo actual no conducirá a una solución, el algoritmo vuelve atrás hasta el padre del nodo para considerar la siguiente opción posible para su último componente; si no existe tal opción, retrocede un nivel más hacia arriba en el árbol, y así sucesivamente. Finalmente, si el algoritmo llega a una solución completa al problema, éste se detendrá (si sólo se requiere una solución) o continuará buscando otras posibles soluciones.

A continuación se presenta un ejemplo que se puede resolver usando backtracking: el problema de las n reinas. El problema consiste en colocar n reinas en un tablero de ajedrez de $n \times n$ de modo que no haya dos reinas que se ataquen entre sí por estar en la misma fila, columna o diagonal.

El algoritmo de backtracking de Gauss y Laquière resuelve el problema de las n reinas realizando una búsqueda en profundidad de todas las soluciones posibles (Erickson, 2019). Las posiciones de las reinas son representadas usando un arreglo $Q[1..n]$, donde $Q[i]$ indica una celda que contiene una reina: $Q[i]$ es la columna de la celda, e i la fila. La variable r representa la fila en la que se encuentra alguna instancia de la recursividad. El índice j representa las columnas, las cuales se recorrerán para verificar si se puede colocar una reina en una determinada posición. En la figura 1.1 se muestra el árbol de recursión completo del algoritmo de Gauss y Laquière cuando $n = 4$.

```
1  Algorithm: PlaceQueens
   Input: Q[1..n], r
   Output: Vacío
2  if r = n + 1 then
3      print Q[1..n];
4  else
5      for j ← 1 to n by 1 do
6          legal ← true;
7          for i ← 1 to r − 1 by 1 do
8              if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i) then
9                  legal ← false;
10             end
11         end
12         if legal = true then
13             Q[r] ← j;
14             PlaceQueens(Q[1..n], r + 1);
15         end
16     end
17 end
```

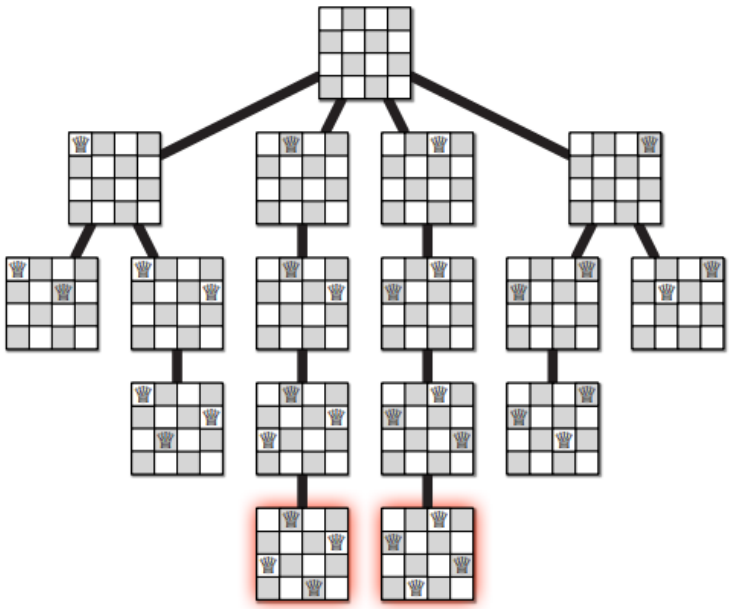


Figura 1.1: Árbol de recursión completo del algoritmo de Gauss y Laquière para el problema de las n reinas, cuando $n = 4$ (Erickson, 2019)

1.3. Distribución normal multivariante

Como ocurre en el caso univariado, existen varios métodos para evaluar la normalidad de los datos multivariados. Sin embargo, a diferencia del análisis univariado, se quiere comprobar el supuesto de normalidad para todas las distribuciones de p dimensiones. Si X es normal multivariante con vector medio μ y matriz de covarianza Σ, escribimos $X \sim N_p(\mu, \Sigma)$. Muchas técnicas estadísticas, como el análisis multivariante de varianza (MANOVA), el análisis de componentes principales (PCA), la correlación canónica, el análisis discriminante y muchas otras, hacen uso de la suposición normal multivariante al hacer inferencias. Además, la mayoría de las teorías en el análisis de datos multivariados se han desarrollado asumiendo normalidad multivariada. Esto se debe a que los procedimientos basados en poblaciones normales son simples y más eficientes, por lo que son muy comunes en las aplicaciones estadísticas (Boakye y Yao, 2016).

Para muestras grandes, se supone que los datos son aproximadamente normales independientemente de la distribución subyacente. Si bien esto es así, la aproximación será mejor cuanto más cercana sea la distribución a la normalidad. Por lo tanto, la detección de desviaciones en la normalidad es crucial.

1.3.1. Tests de normalidad multivariante

Se han propuesto varios tests para evaluar la normalidad multivariante (Korkmaz, Goksuluk, y Zararsiz, 2014). Desafortunadamente, no existe una prueba uniformemente más poderosa conocida y se recomienda realizar varios tests antes de llegar a una conclusión sobre la normalidad. Se recomienda que la elección del método se base en el tipo de desviaciones de la normalidad que se desea investigar según el problema particular en cuestión. Todos los métodos propuestos se pueden agrupar en cuatro categorías principales, a saber: técnicas de bondad de ajuste, procedimientos basados en asimetría y curtosis, tests consistentes e invariantes, y enfoques gráficos y correlacionales.

1.3.2. Algoritmo distribuido para la prueba de normalidad exhaustiva

Un problema que presentan los algoritmos que implementan tests de normalidad multivariante es que se aplican exclusivamente al conjunto de datos completo, excluyendo subconjuntos de variables, lo que aumenta la posibilidad de que falte información sobre las desviaciones de la normalidad dentro de los subespacios de la muestra.

A medida que aumenta el número de variables a considerar, un análisis exhaustivo para cada subconjunto se vuelve poco práctico, ya que tarda demasiado tiempo, más aún teniendo en cuenta que la mayoría de los paquetes estadísticos trabajan en un solo proceso. Una alternativa para disminuir los tiempos totales de ejecución son los algoritmos distribuidos, que a diferencia de los secuenciales, permiten ejecución simultánea de instrucciones. Es por esto que Carvajal y Moreno (2022) diseñaron un algoritmo distribuido para la prueba de normalidad exhaustiva.

Para realizar una prueba de normalidad exhaustiva de una variable m-dimensional se requiere realizar un test por cada subconjunto de variables, por lo que en total se deberán realizar $2^m - 1$ tests.

El algoritmo distribuido utiliza una codificación binaria para representar a cada subconjunto de variables. Por ejemplo, si se tienen 3 variables X_1, X_2, X_3, el subconjunto (X_1, X_3) será representado como 101; y el subconjunto (X_2) será representado como 010. Luego, se reparten los $2^m - 1$ tests entre los distintos procesos para reducir de manera significativa el tiempo total que conlleva realizar una prueba de normalidad exhaustiva. La asignación de tareas de este algoritmo se explica en el capítulo 3.

Capítulo 2

El problema y algoritmos propuestos

2.1. El problema de asignación de tareas

Si una tarea t puede ser representada por su costo computacional como un número entero mayor a cero, se puede definir una lista L que contiene las tareas a realizar por los distintos procesos de un sistema distribuido.

$$L = [t_1, t_2, ..., t_m]$$

Luego, se deben asignar estas tareas a una matriz M, cuyas columnas representan las tareas que cada proceso p deberá realizar.

$$M = \begin{matrix} p_1 & p_2 & \cdots & p_n \end{matrix} \begin{pmatrix} t_{1,1} & t_{1,2} & \cdots & t_{1,n} \\ t_{2,1} & t_{2,2} & \cdots & t_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ t_{r,1} & t_{r,2} & \cdots & t_{r,n} \end{pmatrix}$$

Se define $d_{j,k}$ como la diferencia absoluta entre las sumas de las columnas j y k.

$$d_{j,k} = |\sum_{i=1}^{r} t_{i,j} - \sum_{i=1}^{r} t_{i,k}|$$

Luego, se define D como el conjunto de todas las diferencias entre las columnas.

$$D = \{d_{1,2}, ..., d_{1,n},$$
$$d_{2,3}, ..., d_{2,n},$$
$$d_{n-1,n}\}$$

Así, D tendrá una cantidad de $\frac{1}{2}(n-2)(n-1)$ elementos. Luego, lo que se busca es minimizar el máximo de éstos elementos.

$$\text{Minimizar } z = max(D)$$

Alternativamente, se puede aprovechar el hecho de que la máxima diferencia siempre será la diferencia entre la máxima suma y la mínima suma. Así, el conjunto de todas las sumas se puede definir como S.

$$S = \{\sum_{i=1}^{r} t_{i,1}, ..., \sum_{i=1}^{r} t_{i,n}\}$$

Después, el problema puede definirse como:

$$\text{Minimizar } z = |max(S) - min(S)|$$

2.2. Algoritmos propuestos

Observando la definición del problema, pareciera ser que debe ser resuelto con alguna técnica de programación entera, como “Branch-and-bound”. Sin embargo, no tiene sentido utilizar esta técnica, ya que hacerlo implicaría intentar asignar todos los elementos de la lista de tareas en la primera celda, luego todos los elementos restantes en la segunda celda, y así sucesivamente. Esto es así, debido a que esta técnica recorre el espacio de estados usando “Breadth-first search”. Para el problema de la asignación de tareas, es mucho más razonable utilizar una técnica que recorra el espacio de estados usando “Depth-first search”, como el “backtracking”.
Para la descripción de los algoritmos se utilizará la “programación literaria” propuesta por Donald Knuth (1984), ya que permite explicar la lógica detrás de los algoritmos de una mejor manera que usando sólo pseudocódigo con comentarios.
A continuación se describen las variables comunes de los tres algoritmos:

- arr: un arreglo compuesto de números enteros mayores a cero. Estos números son una representación simplificada de las tareas que el sistema distribuido debe realizar.

- $matrix$: la matriz donde se almacenarán todos los elementos del arreglo arr. Cada columna representa las tareas que cada proceso deberá realizar. La cantidad de filas sólo se sabrá cuando se termina de asignar elementos.
- $ideal$: la suma ideal que cada columna debe tener. Es igual a $total/c$, siendo $total$ la suma de todos los elementos de arr, y c la cantidad de columnas. Se le sumará 1 si es que el resto de la división es mayor a 0.
- f_ideal: tiene el mismo valor que $ideal$, pero no se le suma 1 cuando hay resto en la división $total/c$.
- rem: el resto que resulta de la división $total/c$. Se reparte de manera equitativa en las columnas. De esta forma, rem será igual a la cantidad de columnas que tienen una suma de $f\_ideal + 1$.
- $sums$: un arreglo que contendrá las sumas temporales de las columnas. Así, $sums_j$ será la suma temporal de la columna j en cualquier instante del recorrido de $matrix$.
- rdy: un arreglo de tipo $boolean$ que servirá para identificar a las columnas que ya están listas con la suma ideal. Así, rdy_j será igual a $true$ si es que la columna j está lista con la suma ideal, y será $false$ si no lo está.

2.2.1. Backtracking

Backtrack(arr, $matrix$, $sums$, $size$, c, $ideal$, rem, f_ideal, x, y, $used$, rdy, all_rdy): asigna todos los elementos del arreglo arr en una matriz $matrix$, de forma tal que las sumas de las columnas de $matrix$ tendrán un valor similar. $size$ representa el tamaño de arr, y c representa la cantidad de columnas de $matrix$. arr está compuesto de números enteros mayores a cero, ordenados de menor a mayor. Este algoritmo retorna un valor $boolean$: $true$ si se encontró una solución, y $false$ si no. Esto es así, debido a que se quiere encontrar una sola solución al problema y no todas las soluciones posibles. Los índices x e y representan la fila y la columna de $matrix$ en la que se encuentra una instancia de la recursividad.

Los pasos principales del algoritmo son los siguientes:

1. Verificar que todas las columnas estén listas con la suma ideal.
2. Recorrer arr desde el último elemento hasta el primero.
   a) Asignar un elemento de arr en $matrix$ y actualizar variables.
   b) Llamar a **Backtrack** a la siguiente celda de $matrix$.

c) Remover el número asignado en la celda y las demás actualizaciones de variables.

A continuación se detallan cada uno de los pasos:

1. Verificar que todas las columnas estén listas con la suma ideal.

   Primero, se debe verificar si $all\_rdy = c$, ya que, si es así, se encontró una solución y se debe finalizar el algoritmo. La variable all_rdy es un contador de la cantidad de columnas que están listas con la suma ideal.

   Después, si se encontró una solución, se retornará $true$; y por consiguiente, todas las demás instancias de la recursividad retornarán $true$ también.

2. Recorrer arr desde el último elemento hasta el primero.

   Se recorre arr de esta forma ya que se quiere ir asignando los números de mayor a menor. Esta forma de asignación permite que se pueda buscar por un número más pequeño en los casos donde asignar un número arr_i excedería la suma ideal.

2.a Asignar un elemento de arr en $matrix$ y actualizar variables.

   Antes de asignar un número arr_i en $matrix$, primero se deben verificar tres casos:

   i. Si la columna está lista con la suma ideal, se asigna un 0 a la celda.

   ii. Si el resto rem es igual a 0, y la suma de la columna es igual a f_ideal: se asigna un 0 a la celda, se marca la columna como lista, y se actualiza el contador all_rdy. Este caso se explica mejor con un ejemplo: en el recorrido de las celdas de $matrix$, cuando se visita una celda en la columna y, y se asigna un número arr_i que ocasiona que $arr_i + sums_y = f\_ideal$, la columna no se marcará como lista si es que $rem > 0$. Luego, si todo el resto rem se reparte entre otras columnas en el recorrido de $matrix$, cuando se vuelva a visitar la columna y, ésta debe ser marcada como lista, ya que ahora $ideal = f\_ideal$, pero esto sólo se realizará si es que se verifica por este caso.

   iii. Si $used_i = false$ y $arr_i + sums_y \leq ideal$: se asigna arr_i a la celda, y se actualizan $sums_y$ y $used_i$. $used$ es un arreglo de tipo $boolean$, cuyo elemento $used_i$ será $true$ si es que arr_i ha sido asignado en $matrix$. Además, si $arr_i + sums_y = ideal$, se actualizan rem, rdy y all_rdy.

2.b Llamar a **Backtrack** a la siguiente celda de $matrix$.

   Se llama a una nueva instancia de la recursividad para seguir asignando elementos en $matrix$. Ya que se llama a la siguiente celda, se deben considerar los casos donde $y = c-1$, esto es, cuando y es la última columna de $matrix$; en este caso, se llama a **Backtrack** con valores de x y de y como $x+1$ y 0, respectivamente.

2.c Remover el número asignado en la celda y las demás actualizaciones de variables.

Como en todos los algoritmos del tipo “Backtracking”, se deben remover los cambios realizados en la instancia de la recursividad, cuando se detecta que la configuración actual no conducirá a una posible solución.

Se utiliza una variable rp, a la cual se le asigna el valor de arr_i. Luego, en la siguiente iteración del recorrido de arr, si $arr_i = rp$, se saltará a la siguiente iteración, ya que se sabe que el número rp produce una vuelta atrás.

Complejidad de tiempo del algoritmo: a continuación se presenta la complejidad “Big O” en el peor y en el mejor de los casos.

i. Peor caso: $O(size!)$. Este caso ocurre cuando la mayoría de las veces se remueve una asignación que ya se hizo, esto es, cuando se vuelve atrás en el recorrido de las celdas de $matrix$. La complejidad es factorial ya que se deben realizar todas las permutaciones posibles hasta llegar a una solución.

ii. Mejor caso: $O(size^2)$. Este caso ocurre cuando nunca se hace una vuelta atrás. Cada instancia de la recursividad recorrerá arr desde el final hasta el principio hasta encontrar un número que pueda ser asignado.

2.2.2. Zigzag

Zigzag(arr, $size$, c): asigna todos los elementos del arreglo arr en una matriz $matrix$, de forma tal que las sumas de las columnas de $matrix$ tendrán un valor similar. $size$ representa el tamaño de arr, y c representa la cantidad de columnas de $matrix$. arr está compuesto de números enteros mayores a cero, ordenados de menor a mayor.

Los pasos principales del algoritmo son los siguientes:

1. Recorrer $matrix$ en forma de zigzag.
   a) Asignar los números de mayor a menor.
   b) Si un número no se puede asignar, se busca por un número menor que sí se pueda.

A continuación se detallan cada uno de los pasos:

1. Recorrer $matrix$ en forma de zigzag.

   Se recorre $matrix$ en forma de zigzag: la primera fila, de izquierda a derecha; la segunda fila, de derecha a izquierda; la tercera fila, de izquierda a derecha; y así sucesivamente. Siempre

que se recorra una celda de $matrix$, se le asignará a ésta un elemento de arr, a menos que la celda se encuentre en una columna que ya tiene la suma ideal. En tal caso, se le asignará un 0 a la celda.

```
1  Algorithm: Zigzag
   Input: arr, size, c
   Output: matrix
   // Inicialización de variables...
2  k ← size − 1 ;
3  dir ← false ;
4  for i ← 0 to size − 1 by 1 do
5      if dir = false then
6          for j ← 0 to c − 1 by 1 do
7              AssignNumber(matrix, arr, sums, ideal, f_ideal, i, j, k, rem, rdy)
8          end
9          dir ← true ;
10     else
11         for j ← c − 1 to 0 by −1 do
12             AssignNumber(matrix, arr, sums, ideal, f_ideal, i, j, k, rem, rdy)
13         end
14         dir ← false ;
15     end
16     if k = −1 then
17         break ;
18     end
19 end
20 return matrix ;
```

1.a Asignar los números de mayor a menor.

Se utiliza una variable k que representa el índice de arr que contiene el número que se asignará en $matrix$. Ésta variable k debe ser siempre el mayor índice de arr que tiene un valor de $arr_k \neq 0$. De esta forma, todos los elementos que le siguen al índice k habrán sido asignados en $matrix$, y todos los elementos que anteceden a k no se han asignado, o estarán marcados con un valor de 0, indicando que ya se han asignado. Esta variable k se inicializa con el último índice de arr, e irá disminuyendo hasta llegar al primer índice de arr. De esta forma, arr será recorrido desde el final hasta el principio. Se recorre de esta

forma ya que arr está ordenado de menor a mayor. Entonces, los números mayores serán asignados primero en $matrix$.

Antes de asignar un número, primero se debe observar el valor del resto, rem, para asignar a $ideal$ el valor adecuado. Luego se debe verificar si $k = -1$, ya que si lo es, eso significa que todos los números ya se han asignado en $matrix$, por lo que se debe finalizar el algoritmo.

```
1  Algorithm: AssignNumber
   Input: matrix, arr, sums, ideal, f_ideal, i, j, k, rem, rdy
   Output: Vacío
2  if rem ≤ 0 then
3  |  ideal ← f_ideal;
4  end
5  if k = −1 then
6  |  return;
7  end
   // k debe ser el mayor índice de arr que tiene un valor de arr_k ≠ 0
8  if arr_k = 0 then
9  |  for z ← k − 1 to 0 by −1 do
10 |  |  if arr_z ≠ 0 then
11 |  |  |  k ← z;
12 |  |  |  break ;
13 |  |  end
14 |  end
15 end
```

Luego, se distinguen cuatro condiciones.

i. Si la columna j está lista con la suma ideal, se asigna un cero a la celda.

ii. Si ya no hay resto y la suma de la columna j es igual a la suma ideal (sin el resto), pero aún así la columna j no está marcada como lista. Esto ocurre debido a que el valor de $ideal$ cambia según el valor de rem. Por ejemplo, si en el recorrido de $matrix$, en un instante se deja la columna j con una suma de f_ideal y todavía queda resto, la columna j no será marcada como lista; luego, cuando todo el resto se reparta entre otras columnas y se vuelva a visitar la columna j, se debe marcar la columna como lista, ya que ahora que no hay resto, el valor de $ideal$ será igual al de f_ideal.

iii. Si el número arr_k puede ser asignado en la celda, se actualizan los valores de las variables

que sean necesarias.

iv. Si el número arr_k no puede ser asignado a la celda, se busca por un número menor que sí se pueda. Este caso se detalla más en el paso principal (1.b) del algoritmo.

```
// continuación de AssignNumber (1)
16 if rdy_j = true then
17 |   matrix_{i,j} ← 0 ;
18 end
19 else if rem ≤ 0 and sums_j = f_ideal then
20 |   rdy_j ← true ;
21 |   matrix_{i,j} ← 0 ;
22 end
23 else if sums_j + arr_k ≤ ideal then
24 |   if sums_j + arr_k = ideal then
25 |   |   rdy_j ← true ;
26 |   |   if rem > 0 then
27 |   |   |   rem ← rem − 1 ;
28 |   |   end
29 |   end
30 |   matrix_{i,j} ← arr_k ;
31 |   sums_j ← sums_j + matrix_{i,j} ;
32 |   k ← k − 1 ;
33 end
```

1.b Si un número no se puede asignar, se busca por un número menor que sí se pueda.

Ya que los números mayores serán asignados primero en $matrix$, cuando se quiera asignar un número arr_k que ocasione que $arr_k + sums_j > ideal$, simplemente se busca por un número menor a arr_k en los elementos de arr que anteceden a arr_k.

Si no se logra encontrar un número num tal que $num + sums_j \leq ideal$, se asignará el menor número encontrado, ya que éste tendrá el menor impacto en la suma de la columna.

```
// continuación de AssignNumber (2)
else
    num_found ← false ;
    min_num ← ∞ ;
    for q ← k − 1 to 0 by −1 do
        if arr_q ≠ 0 and arr_q < min_num then
            min_num ← arr_q ;
            min_q ← q ;
        end
        if arr_q ≠ 0 and sums_j + arr_q ≤ ideal then
            if sums_j + arr_q = ideal then
                rdy_j ← true ;
                if rem > 0 then
                    rem ← rem − 1 ;
                end
            end
            matrix_{i,j} ← arr_q ;
            sums_j ← sums_j + matrix_{i,j} ;
            arr_q ← 0 ;
            num_found ← true ;
            break ;
        end
    end
    if num_found = false then
        rdy_j ← true ;
        matrix_{i,j} ← arr_{min_q} ;
        sums_j ← sums_j + matrix_{i,j} ;
        arr_{min_q} ← 0 ;
        if rem > 0 then
            rem ← rem − 1 − (sums_j − ideal) ;
        end
    end
end
```

Complejidad de tiempo del algoritmo: Si bien, el ciclo principal del algoritmo es el de recorrer $matrix$, esto sólo se hace así por simplicidad. El verdadero ciclo principal es el de recorrer arr

al ir asignando sus elementos, y, cuando se terminan de asignar todos, se rompe el recorrido de $matrix$. Entonces, la complejidad de tiempo se mide con respecto al tamaño de arr, esto es, $size$. A continuación se presenta la complejidad "Big O" en el peor y en el mejor de los casos.

i. Peor caso: $O(size^2)$. Este caso ocurre cuando la mayoría de las veces se debe buscar un número menor al número representado por el índice k.

ii. Mejor caso: $O(size)$. Este caso ocurre cuando nunca es necesario buscar un número menor al número representado por el índice k. Cada elemento de arr se visita sólo una vez.

2.2.3. Columns First

ColFirst(arr, $size$, c): asigna todos los elementos del arreglo arr en una matriz $matrix$, de forma tal que las sumas de las columnas de $matrix$ tendrán un valor similar. $size$ representa el tamaño de arr, y c representa la cantidad de columnas de $matrix$. arr está compuesto de números enteros mayores a cero, ordenados de menor a mayor.

Los pasos principales del algoritmo son los siguientes:

1. Recorrer $matrix$ por sus columnas.

    a) Asignar los números de mayor a menor.

    b) Si un número no se puede asignar, se busca por un número menor que sí se pueda.

Este algoritmo utiliza la misma lógica que el algoritmo Zigzag para la asignación de números en $matrix$, esto es, los pasos principales (1.a) y (1.b); pero se distingue de Zigzag en el recorrido de $matrix$, el cual será por sus columnas. Entonces, cada columna se dejará lista con la suma ideal y no se volverá a visitar en el recorrido de $matrix$. Debido a esto, $sums$ ya no será un arreglo, sino que un solo número que se reiniciará con un valor de 0 cada vez que se cambie de columna en el recorrido de $matrix$.
Debido a que los números mayores son asignados primero en $matrix$, las primeras columnas contendrán los números mayores de arr, mientras que las últimas columnas contendrán los números menores de arr. Esto no sucede en el algoritmo de Zigzag.
Complejidad de tiempo del algoritmo: Al igual que el algoritmo de Zigzag, el ciclo principal de este algoritmo es el de recorrer $matrix$, pero esto sólo se hace así por simplicidad. El verdadero ciclo principal es el de recorrer arr al ir asignando sus elementos. Luego, se rompe el recorrido de $matrix$ una vez que se asignan todos los elementos. Entonces, la complejidad de tiempo se mide con respecto al tamaño de arr, esto es, $size$. A continuación se presenta la complejidad "Big O" en el peor y en el mejor de los casos.

i. Peor caso: $O(size^2)$. Este caso ocurre cuando la mayoría de las veces se debe buscar un número menor al número representado por el índice k.

ii. Mejor caso: $O(size)$. Este caso ocurre cuando nunca es necesario buscar un número menor al número representado por el índice k. Cada elemento de arr se visita sólo una vez.

Capítulo 3

Aplicaciones y experimentos

3.1. Aplicaciones

La mejora en la asignación de tareas puede ser usada por cualquier sistema distribuido en donde el costo computacional de cada tarea se pueda representar como un número entero mayor a cero. En este trabajo, se aplicó la mejora en la asignación de tareas en el algoritmo distribuido para la prueba de normalidad exhaustiva. A continuación se explica cómo se realizaba la asignación de tareas originalmente en este algoritmo, y cómo se logró mejorar.

En el algoritmo distribuido para la prueba de normalidad exhaustiva, se tienen un total de $2^m - 1$ tareas a realizar, siendo m la cantidad de variables a considerar. Para caracterizar de manera simplificada el costo computacional de realizar cada una de las $2^m - 1$ tareas a resolver, se puede utilizar el orden de la matriz de covarianzas **S** asociada a cada tarea, ya que la operación mas costosa a realizar es el cálculo de su matriz inversa.

Para explicar de mejor forma la asignación de tareas, se presenta un ejemplo: resolver el problema con cuatro variables, lo cual significa resolver 15 tareas. Las tablas 3.1 y 3.2 muestran la asignación de tareas usando dos y cuatro nodos respectivamente. Cada columna contiene dos números: el primero corresponde a la representación decimal de la tarea a resolver, y el número entre paréntesis, el numero de variables a considerar; nótese que éste último número corresponde a la dimensión de la matriz **S**, lo que significa que es el número que representa el costo computacional de realizar la tarea.

Tabla 3.1: Asignación original
con 2 nodos (procesos)

Nodo 0		Nodo 1	
1	(1)	9	(2)
2	(1)	10	(2)
3	(2)	11	(3)
4	(1)	12	(2)
5	(2)	13	(3)
6	(2)	14	(3)
7	(3)	15	(4)
8	(1)		
	13		19

Tabla 3.2: Asignación original con 4 nodos (procesos)

Nodo 0		Nodo 1		Nodo 2		Nodo 3	
1	(1)	5	(2)	9	(2)	13	(3)
2	(1)	6	(2)	10	(2)	14	(3)
3	(2)	7	(3)	11	(3)	15	(4)
4	(1)	8	(1)	12	(2)		
	5		8		9		10

El ultimo valor de cada columna representa la suma de los números entre paréntesis y por tanto una representación del costo total que cada nodo realizará. Se puede observar que en el caso de utilizar dos nodos, hay una diferencia de seis unidades entre los costos de ambos nodos; y en el caso de la asignación con cuatro nodos, la diferencia entre el valor mínimo y máximo es de cinco unidades.

Dado que siempre se conoce el orden de las $2^m - 1$ matrices a invertir y el numero de nodos a utilizar, una asignación óptima de las tareas a realizar consiste en distribuir entre los nodos, los $2^m - 1$ valores de modo tal que las diferencias de las sumas de las tareas entre los nodos sea la mínima posible. En las tablas 3.3 y 3.4 se muestra una asignación óptima para el ejemplo anterior. Para lograr esta asignación mejorada, se utilizó el algoritmo de Zigzag explicado en el capítulo 2.

Tabla 3.3: Asignación mejorada con 2 nodos

Nodo 0		Nodo 1	
15	(4)	7	(3)
11	(3)	13	(3)
14	(3)	3	(2)
5	(2)	6	(2)
9	(2)	10	(2)
1	(1)	12	(2)
2	(1)	4	(1)
		8	(1)
	16		16

Tabla 3.4: Asignación mejorada con 4 nodos

Nodo 0		Nodo 1		Nodo 2		Nodo 3	
15	(4)	7	(3)	11	(3)	13	(3)
3	(2)	5	(2)	6	(2)	14	(3)
9	(2)	10	(2)	12	(2)	1	(1)
		2	(1)	4	(1)	8	(1)
	8		8		8		8

En la implementación de los algoritmos propuestos, cada elemento del arreglo arr y la matriz $matrix$, es una estructura de datos que contiene dos números enteros: la cantidad de 1s activos en las combinaciones de las m variables y el índice específico para formar una combinación de variables. Este último número lo utiliza el programa que implementa el algoritmo distribuido para la prueba de normalidad exhaustiva.

3.2. Experimentos

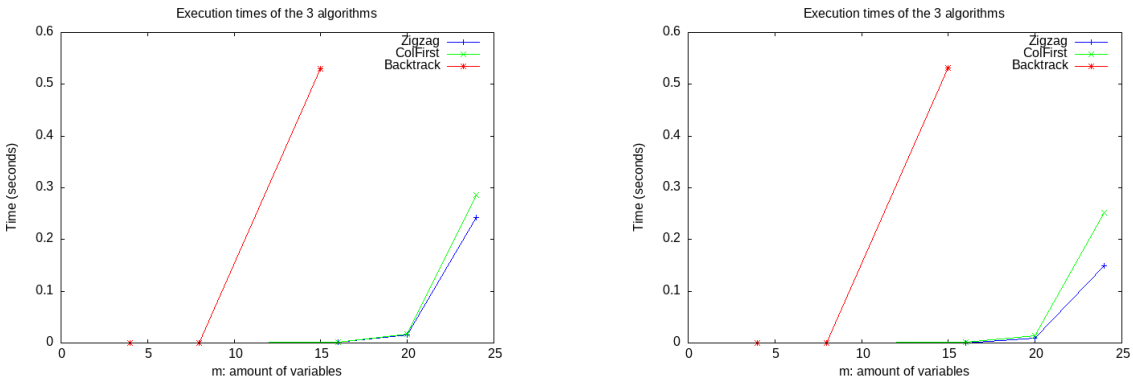
En la tabla 3.5 se detallan las características del *cluster* en donde se realizaron los experimentos del algoritmo distribuido para la prueba de normalidad exhaustiva. Para los experimentos de los tres algoritmos propuestos se utilizó un solo computador del *cluster*.

Tabla 3.5: Características del *cluster*

Procesador	Intel Core i7-4790 CPU @ 3.60 Ghz x 8
Memoria RAM	6.4 Gib
Sistema operativo	Ubuntu 18.04 LTS - 64 bits
Lenguaje de programación	Lenguaje C. Compilador: gcc

3.2.1. Experimentos de los tres algoritmos propuestos

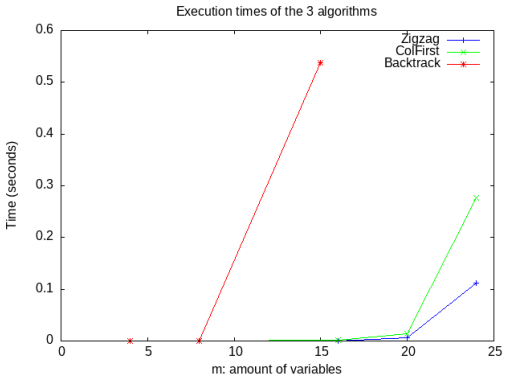
En las figuras 3.1a, 3.1b, 3.2a, 3.2b y 3.3 se muestran los tiempos de ejecución de los tres algoritmos propuestos según la cantidad de variables m, y variando la cantidad de procesos. Los algoritmos son secuenciales, sólo se especifica la cantidad de procesos para la construcción de la matriz.



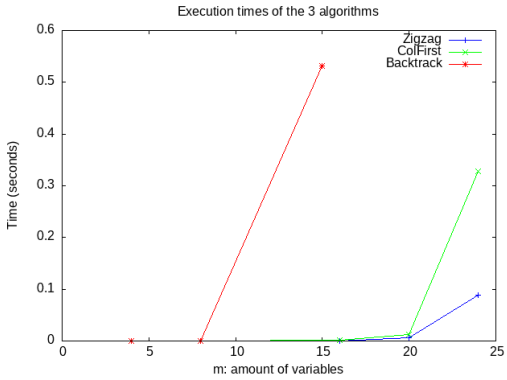
(a) Tiempos de ejecución con 2 procesos

(b) Tiempos de ejecución con 4 procesos

Figura 3.1: Tiempos de ejecución de los 3 algoritmos - 2 y 4 procesos



(a) Tiempos de ejecución con 8 procesos



(b) Tiempos de ejecución con 16 procesos

Figura 3.2: Tiempos de ejecución de los 3 algoritmos - 8 y 16 procesos

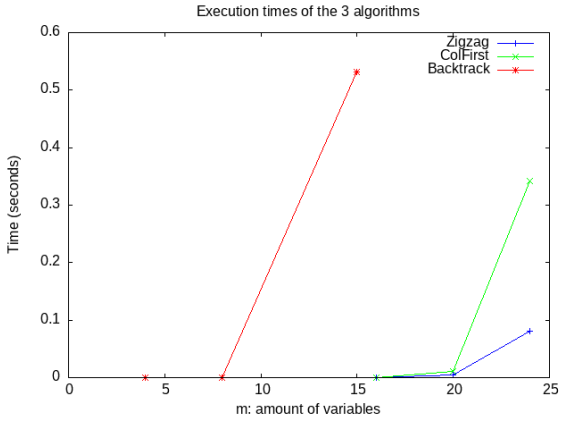


Figura 3.3: Tiempos de ejecución de los 3 algoritmos - 32 procesos

Se puede observar que el algoritmo que produjo mejores resultados fue el de Zigzag. Además, en el desarrollo de los experimentos de este algoritmo, al observar la matriz resultante, se pudo notar que sólo en las últimas filas de la matriz fue necesario realizar el paso principal (1.b), esto es, buscar por un número menor si es que el número representado por el índice k no se puede asignar. Esto muestra que en la práctica, la complejidad de tiempo del algoritmo es muy cercana a $O(size)$, lo cual se refleja claramente en los tiempos de ejecución. Lo mismo se aplica para el algoritmo ColFirst. Algo parecido ocurre en el algoritmo de Backtracking, en donde sólo en las últimas filas fue necesario hacer vueltas hacia atrás, por lo que la complejidad resulta mucho más cercana a $O(size^2)$ que a $O(size!)$.

Con respecto a la matriz resultante de los algoritmos, se pudo observar que ColFirst produjo matrices con una gran cantidad de ceros, lo cual significa que hay procesos que realizan muchas más tareas que otros procesos. Esto no es deseable debido a que los números son representaciones simplificadas de las tareas que los procesos deben realizar. Aparte del cálculo de la subtarea que

posee la mayor complejidad de tiempo, habrá otras subtareas que también demandan tiempo de proceso, las cuales se harán notar mientras mayor sea la cantidad de tareas que un proceso deba realizar. En este aspecto, los algoritmos Zigzag y Backtracking produjeron matrices con una cantidad de ceros insignificante, lo cual es más deseable. La matriz M_1 muestra un esquema de una matriz producida con el algoritmo ColFirst, mientras que la matriz M_2 muestra un esquema de una matriz producida con el algoritmo Zigzag o con el algoritmo de Backtracking.

$$M_1 = \begin{array}{c} \begin{matrix} p_1 & p_2 & p_3 & \cdots & p_n \end{matrix} \\ \begin{pmatrix} t_{1,1} & t_{1,2} & t_{1,3} & \cdots & t_{1,n} \\ t_{2,1} & t_{2,2} & t_{2,3} & \cdots & t_{2,n} \\ 0 & t_{3,2} & t_{3,3} & \cdots & t_{3,n} \\ 0 & 0 & t_{4,3} & \cdots & t_{4,n} \\ 0 & 0 & 0 & \cdots & t_{5,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & t_{r,n} \end{pmatrix} \end{array}$$

$$M_2 = \begin{array}{c} \begin{matrix} p_1 & p_2 & p_3 & \cdots & p_n \end{matrix} \\ \begin{pmatrix} t_{1,1} & t_{1,2} & t_{1,3} & \cdots & t_{1,n} \\ t_{2,1} & t_{2,2} & t_{2,3} & \cdots & t_{2,n} \\ t_{3,1} & t_{3,2} & t_{3,3} & \cdots & t_{3,n} \\ t_{4,1} & t_{4,2} & t_{4,3} & \cdots & t_{4,n} \\ t_{5,1} & t_{5,2} & t_{5,3} & \cdots & t_{5,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_{r,1} & t_{r,2} & t_{r,3} & \cdots & t_{r,n} \end{pmatrix} \end{array}$$

3.2.2. Experimentos del algoritmo distribuido para la prueba de normalidad exhaustiva

A continuación se presentan los experimentos realizados con la asignación de tareas original que tenía el algoritmo, y con asignación de tareas mejorada.

Experimentos con asignación original

Desde la tabla 3.6 hasta la 3.13 se muestra el tiempo de ejecución (en segundos) variando la cantidad de nodos, observaciones y variables.

Tabla 3.6: Tiempo de ejecución (en segundos) - 100,000 Observaciones - 4 Variables

100,000 Observaciones - 4 Variables														
Cantidad de nodos														
2		4		8				15						
[0]	0.45	[0]	0.12	[0]	0.009	[1]	0.11	[0]	0.009	[1]	0.01	[2]	0.11
[1]	0.76	[1]	0.33	[2]	0.12	[3]	0.22	[3]	0.009	[4]	0.11	[5]	0.11
		[2]	0.36	[4]	0.11	[5]	0.22	[6]	0.11	[7]	0.01	[8]	0.11
		[3]	0.45	[6]	0.22	[7]	0.22	[9]	0.12	[10]	0.11	[11]	0.11
								[12]	0.11	[13]	0.11	[14]	0.11

Tabla 3.7: Tiempo de ejecución (en segundos) - 100,000 Observaciones - 8 Variables

100,000 Observaciones - 8 Variables											
Cantidad de nodos											
2		4		8		16		32			
[0]	13.24	[0]	6.28	[0]	2.87	[0]	1.34	[0]	0.55	[1]	0.90
[1]	14.00	[1]	6.91	[1]	3.38	[1]	1.75	[2]	1.02	[3]	1.05
		[2]	7.56	[2]	3.70	[2]	1.96	[4]	0.92	[5]	1.16
		[3]	7.38	[3]	3.68	[3]	1.99	[6]	1.14	[7]	1.06
				[4]	3.37	[4]	1.79	[8]	0.93	[9]	1.06
				[5]	3.59	[5]	1.87	[10]	1.24	[11]	1.12
				[6]	3.69	[6]	1.95	[12]	1.04	[13]	1.19
				[7]	3.65	[7]	1.93	[14]	1.07	[15]	1.11
						[8]	1.77	[16]	0.99	[17]	1.07
						[9]	1.89	[18]	1.22	[19]	1.12
						[10]	2.13	[20]	1.17	[21]	1.12
						[11]	2.01	[22]	1.18	[23]	1.19
						[12]	1.92	[24]	1.10	[25]	1.17
						[13]	1.90	[26]	1.11	[27]	1.13
						[14]	1.98	[28]	1.04	[29]	1.17
						[15]	1.96	[30]	1.06	[31]	1.09

Tabla 3.8: Tiempo de ejecución (en segundos) - 100,000 Observaciones - 16 Variables

100,000 Observaciones - 16 Variables												
Cantidad de nodos												
2		4		8		16		32				
[0]	3,881.07	[0]	1,914.04	[0]	948.16	[0]	519.04	[0]	320.13	[1]	331.60
[1]	3,940.48	[1]	1,947.48	[1]	962.66	[1]	527.05	[2]	353.58	[3]	349.34
		[2]	2,127.23	[2]	1,053.31	[2]	572.40	[4]	330.86	[5]	341.72
		[3]	2,040.02	[3]	1,011.26	[3]	551.82	[6]	348.28	[7]	356.45
				[4]	959.65	[4]	524.71	[8]	329.56	[9]	340.75
				[5]	990.75	[5]	539.58	[10]	363.82	[11]	358.66
				[6]	1,010.37	[6]	548.89	[12]	339.81	[13]	353.07
				[7]	1,017.84	[7]	553.43	[14]	359.19	[15]	368.31
						[8]	532.23	[16]	329.97	[17]	340.72
						[9]	541.01	[18]	364.39	[19]	360.28
						[10]	584.04	[20]	339.02	[21]	353.32
						[11]	565.99	[22]	358.88	[23]	368.76
						[12]	540.80	[24]	337.38	[25]	350.13
						[13]	552.00	[26]	374.82	[27]	370.44
						[14]	563.25	[28]	349.73	[29]	364.09
						[15]	567.62	[30]	370.73	[31]	379.29

Tabla 3.9: Tiempo de ejecución (en segundos) - 100,000 Observaciones - 20 Variables

100,000 Observaciones - 20 Variables												
Cantidad de nodos												
2		4		8		16		32				
[0]	67,050.41	[0]	33,067.56	[0]	16,307.53	[0]	9,096.10	[0]	5,892.93	[1]	6,204.45
[1]	68,652.99	[1]	33,847.46	[1]	16,662.03	[1]	9,340.89	[2]	6,556.14	[3]	6,649.54
		[2]	36,779.37	[2]	18,270.51	[2]	10,085.90	[4]	6,189.88	[5]	6,542.41
		[3]	35,640.14	[3]	17,597.75	[3]	9,891.57	[6]	6,621.75	[7]	6,957.73
				[4]	16,606.13	[4]	9,352.53	[8]	6,095.29	[9]	6,443.04
				[5]	17,262.62	[5]	9,685.10	[10]	6,802.95	[11]	6,906.13
				[6]	17,621.63	[6]	9,831.55	[12]	6,426.19	[13]	6,800.29
				[7]	17,828.18	[7]	10,085.03	[14]	6,871.09	[15]	7,243.39
						[8]	9,338.51	[16]	6,092.70	[17]	6,424.53
						[9]	9,583.50	[18]	6,801.60	[19]	6,915.35
						[10]	10,405.46	[20]	6,431.43	[21]	6,800.08
						[11]	10,210.16	[22]	6,868.96	[23]	7,243.92
						[12]	9,604.65	[24]	6,272.86	[25]	6,661.84
						[13]	9,952.13	[26]	7,023.59	[27]	7,134.93
						[14]	10,168.74	[28]	6,647.99	[29]	7,017.65
						[15]	10,418.89	[30]	7,102.40	[31]	7,488.99

Tabla 3.10: Tiempo de ejecución (en segundos) - 200,000 Observaciones - 4 Variables

200,000 Observaciones - 4 Variables															
Cantidad de nodos															
2		4		8				15							
[0]	0.52	[0]	0.15	[0]	0.02	[1]	0.13	[0]	0.02	[1]	0.02	[2]	0.12
[1]	0.83	[1]	0.36	[2]	0.14	[3]	0.25	[3]	0.02	[4]	0.12	[5]	0.12
		[2]	0.40	[4]	0.13	[5]	0.25	[6]	0.13	[7]	0.02	[8]	0.12
		[3]	0.51	[6]	0.25	[7]	0.25	[9]	0.14	[10]	0.13	[11]	0.12
								[12]	0.12	[13]	0.13	[14]	0.13

Tabla 3.11: Tiempo de ejecución (en segundos) - 200,000 Observaciones - 8 Variables

200,000 Observaciones - 8 Variables												
Cantidad de nodos												
2		4		8		16		32				
[0]	14.48	[0]	6.87	[0]	3.15	[0]	1.51	[0]	0.62	[1]	1.03
[1]	15.48	[1]	7.56	[1]	3.70	[1]	1.88	[2]	1.09	[3]	1.24
		[2]	8.32	[2]	4.06	[2]	2.05	[4]	0.99	[5]	1.12
		[3]	8.27	[3]	4.09	[3]	2.23	[6]	1.16	[7]	1.18
				[4]	3.70	[4]	1.90	[8]	1.02	[9]	1.17
				[5]	4.11	[5]	2.09	[10]	1.24	[11]	1.21
				[6]	4.09	[6]	2.14	[12]	1.30	[13]	1.32
				[7]	4.20	[7]	2.20	[14]	1.23	[15]	1.21
						[8]	1.92	[16]	0.99	[17]	1.20
						[9]	2.15	[18]	1.35	[19]	1.23
						[10]	2.22	[20]	1.19	[21]	1.25
						[11]	2.20	[22]	1.27	[23]	1.20
						[12]	2.11	[24]	1.22	[25]	1.17
						[13]	2.11	[26]	1.35	[27]	1.25
						[14]	2.17	[28]	1.16	[29]	1.26
						[15]	2.17	[30]	1.32	[31]	1.19

Tabla 3.12: Tiempo de ejecución (en segundos) - 200,000 Observaciones - 16 Variables

200,000 Observaciones - 16 Variables												
Cantidad de nodos												
2		4		8		16		32				
[0]	4,701.15	[0]	2,290.69	[0]	1,120.45	[0]	608.20	[0]	386.95	[1]	413.77
[1]	4,899.70	[1]	2,388.48	[1]	1,165.60	[1]	636.66	[2]	437.11	[3]	454.63
		[2]	2,605.56	[2]	1,271.58	[2]	687.83	[4]	412.76	[5]	449.60
		[3]	2,583.71	[3]	1,268.44	[3]	688.47	[6]	453.49	[7]	487.16
				[4]	1,166.94	[4]	638.95	[8]	407.73	[9]	439.77
				[5]	1,235.06	[5]	673.24	[10]	463.61	[11]	480.96
				[6]	1,264.38	[6]	686.73	[12]	441.04	[13]	474.66
				[7]	1,304.25	[7]	717.10	[14]	481.63	[15]	518.46
						[8]	634.67	[16]	407.61	[17]	439.71
						[9]	668.82	[18]	462.19	[19]	481.15
						[10]	722.65	[20]	439.63	[21]	473.76
						[11]	721.30	[22]	480.43	[23]	518.47
						[12]	670.60	[24]	429.46	[25]	458.96
						[13]	706.46	[26]	484.26	[27]	503.76
						[14]	719.15	[28]	461.21	[29]	498.42
						[15]	751.32	[30]	503.67	[31]	542.12

Tabla 3.13: Tiempo de ejecución (en segundos) - 200,000 Observaciones - 20 Variables

200,000 Observaciones - 20 Variables												
Cantidad de nodos												
2		4		8		16		32				
[0]	87,145.64	[0]	42,121.64	[0]	20,553.54	[0]	11,527.99	[0]	7,876.15	[1]	8,704.83
[1]	91,712.49	[1]	44,747.11	[1]	21,675.29	[1]	12,297.63	[2]	9,017.12	[3]	9,687.77
		[2]	48,329.06	[2]	23,461.13	[2]	13,109.88	[4]	8,683.53	[5]	9,554.31
		[3]	48,848.29	[3]	23,427.92	[3]	13,405.69	[6]	9,637.80	[7]	10,592.73
				[4]	21,663.28	[4]	12,293.67	[8]	8,415.71	[9]	9,275.62
				[5]	23,242.05	[5]	13,186.53	[10]	9,602.51	[11]	10,321.40
				[6]	23,529.08	[6]	13,390.62	[12]	9,287.94	[13]	10,226.53
				[7]	24,608.18	[7]	14,190.07	[14]	10,290.35	[15]	11,312.10
						[8]	12,194.47	[16]	8,394.94	[17]	9,277.15
						[9]	13,026.82	[18]	9,604.87	[19]	10,324.11
						[10]	13,864.24	[20]	9,289.53	[21]	10,227.42
						[11]	14,185.73	[22]	10,301.52	[23]	11,332.35
						[12]	13,029.99	[24]	8,835.08	[25]	9,747.83
						[13]	13,997.27	[26]	10,097.95	[27]	10,842.45
						[14]	14,212.58	[28]	9,778.14	[29]	10,733.10
						[15]	15,062.83	[30]	10,830.85	[31]	11,861.18

Al observar los tiempos de ejecución de los experimentos se pueden distinguir diferencias significativas en los distintos nodos. Por ejemplo, en el caso de procesar 200,000 observaciones y 20 variables, utilizando 32 procesos, el menor tiempo de procesamiento corresponde al nodo cero (7,876.51 segundos) y el mayor tiempo de procesamiento al nodo 31 (11,861.18 segundos). Esto significa que la mayor diferencia de tiempo es de 3,984.67 segundos, lo que significa que no se tiene un buen balance de carga. Esto conlleva a que los demás procesos deberán esperar a que el nodo 31 termine para que se finalice todo el sistema distribuido.

A continuación se muestran los experimentos con una asignación de tareas mejorada, la cual se consiguió con el algoritmo de Zigzag.

Experimentos con asignación mejorada

Desde la tabla 3.14 hasta la 3.21 se muestra el tiempo de ejecución (en segundos) variando la cantidad de nodos, observaciones y variables.

Tabla 3.14: Tiempo de ejecución (en segundos) - 100,000 Observaciones - 4 Variables

100,000 Observaciones - 4 Variables														
Cantidad de nodos														
2		4		8				15						
[0]	0.68	[0]	0.24	[0]	0.22	[1]	0.11	[0]	0.11	[1]	0.11	[2]	0.11
[1]	0.58	[1]	0.33	[2]	0.12	[3]	0.12	[3]	0.11	[4]	0.11	[5]	0.11
		[2]	0.34	[4]	0.12	[5]	0.12	[6]	0.11	[7]	0.11	[8]	0.14
		[3]	0.34	[6]	0.22	[7]	0.22	[9]	0.009	[10]	0.009	[11]	0.009
								[12]	0.009	[13]	0.11	[14]	0.11

Tabla 3.15: Tiempo de ejecución (en segundos) - 100,000 Observaciones - 8 Variables

100,000 Observaciones - 8 Variables																
Cantidad de nodos																
2		4		8		16				32						
[0]	14.29	[0]	7.08	[0]	3.43	[0]	1.95			[0]	1.09	[1]	1.00			
[1]	13.83	[1]	6.93	[1]	3.51	[1]	1.82			[2]	0.95	[3]	1.01			
		[2]	7.04	[2]	3.52	[2]	1.82			[4]	0.92	[5]	0.95			
		[3]	6.98	[3]	3.54	[3]	1.86			[6]	1.00	[7]	1.00			
				[4]	3.53	[4]	1.87			[8]	0.97	[9]	1.00			
				[5]	3.52	[5]	1.83			[10]	1.07	[11]	1.09			
				[6]	3.50	[6]	1.82			[12]	1.08	[13]	1.17			
				[7]	3.52	[7]	1.84			[14]	1.05	[15]	1.17			
						[8]	1.86			[16]	1.08	[17]	1.06			
						[9]	1.81			[18]	1.17	[19]	1.07			
						[10]	1.90			[20]	1.07	[21]	1.08			
						[11]	1.94			[22]	1.06	[23]	1.12			
						[12]	1.95			[24]	1.10	[25]	1.16			
						[13]	1.92			[26]	1.05	[27]	1.06			
						[14]	1.91			[28]	1.07	[29]	1.08			
						[15]	1.92			[30]	1.05	[31]	1.06			

Tabla 3.16: Tiempo de ejecución (en segundos) - 100,000 Observaciones - 16 Variables

100,000 Observaciones - 16 Variables												
Cantidad de nodos												
2		4		8		16		32				
[0]	3,958.30	[0]	1,976.73	[0]	990.43	[0]	543.32	[0]	350.04	[1]	349.97
[1]	3,938.44	[1]	1,977.01	[1]	989.50	[1]	540.87	[2]	351.79	[3]	349.86
		[2]	1,984.22	[2]	994.80	[2]	543.47	[4]	350.21	[5]	347.84
		[3]	1,977.36	[3]	990.34	[3]	543.83	[6]	348.62	[7]	350.71
				[4]	989.86	[4]	541.16	[8]	351.00	[9]	349.61
				[5]	990.30	[5]	543.15	[10]	349.32	[11]	349.63
				[6]	989.52	[6]	539.85	[12]	349.49	[13]	347.22
				[7]	990.59	[7]	542.52	[14]	346.44	[15]	349.56
						[8]	544.72	[16]	349.22	[17]	351.16
						[9]	540.11	[18]	351.57	[19]	350.02
						[10]	542.35	[20]	349.22	[21]	348.05
						[11]	543.06	[22]	347.34	[23]	348.10
						[12]	540.17	[24]	349.69	[25]	350.80
						[13]	542.71	[26]	350.11	[27]	350.79
						[14]	538.52	[28]	351.33	[29]	349.15
						[15]	541.97	[30]	348.00	[31]	347.75

Tabla 3.17: Tiempo de ejecución (en segundos) - 100,000 Observaciones - 20 Variables

100,000 Observaciones - 20 Variables												
Cantidad de nodos												
2		4		8		16		32				
[0]	67,762.78	[0]	34,266.23	[0]	16,987.92	[0]	9,615.08	[0]	6,701.43	[1]	6,691.62
[1]	66,075.38	[1]	34,196.03	[1]	17,087.07	[1]	9,503.30	[2]	6,692.18	[3]	6,696.57
		[2]	33,794.97	[2]	17,070.52	[2]	9,508.17	[4]	6,676.47	[5]	6,670.55
		[3]	33,087.09	[3]	16,804.89	[3]	9,542.89	[6]	6,651.73	[7]	6,660.63
				[4]	16,533.16	[4]	9,598.12	[8]	6,669.66	[9]	6,688.70
				[5]	16,526.96	[5]	9,500.61	[10]	6,693.40	[11]	6,701.27
				[6]	16,556.03	[6]	9,494.76	[12]	6,672.07	[13]	6,670.01
				[7]	16,536.44	[7]	9,518.90	[14]	6,651.11	[15]	6,656.01
						[8]	9,616.19	[16]	6,657.44	[17]	6,686.57
						[9]	9,499.68	[18]	6,695.05	[19]	6,692.29
						[10]	9,507.08	[20]	6,683.40	[21]	6,666.15
						[11]	9,542.26	[22]	6,649.78	[23]	6,668.23
						[12]	9,597.02	[24]	6,667.01	[25]	6,685.52
						[13]	9,500.58	[26]	6,699.73	[27]	6,691.29
						[14]	9,494.03	[28]	6,681.19	[29]	6,672.06
						[15]	9,517.83	[30]	6,653.84	[31]	6,673.74

Tabla 3.18: Tiempo de ejecución (en segundos) - 200,000 Observaciones - 4 Variables

200,000 Observaciones - 4 Variables															
Cantidad de nodos															
2		4		8				15							
[0]	0.78	[0]	0.29	[0]	0.24	[1]	0.12	[0]	0.12	[1]	0.12	[2]	0.13
[1]	0.66	[1]	0.37	[2]	0.14	[3]	0.14	[3]	0.12	[4]	0.12	[5]	0.12
		[2]	0.39	[4]	0.14	[5]	0.14	[6]	0.12	[7]	0.12	[8]	0.12
		[3]	0.39	[6]	0.24	[7]	0.24	[9]	0.02	[10]	0.02	[11]	0.02
								[12]	0.02	[13]	0.13	[14]	0.12

Tabla 3.19: Tiempo de ejecución (en segundos) - 200,000 Observaciones - 8 Variables

200,000 Observaciones - 8 Variables												
Cantidad de nodos												
2		4		8		16		32				
[0]	15.63	[0]	7.85	[0]	3.84	[0]	2.10	[0]	1.18	[1]	1.18
[1]	15.60	[1]	7.77	[1]	3.93	[1]	2.03	[2]	1.07	[3]	1.09
		[2]	7.92	[2]	3.96	[2]	2.09	[4]	1.07	[5]	1.16
		[3]	7.92	[3]	3.97	[3]	2.05	[6]	1.04	[7]	1.14
				[4]	3.98	[4]	2.06	[8]	1.13	[9]	0.99
				[5]	3.97	[5]	2.05	[10]	1.20	[11]	1.16
				[6]	3.95	[6]	2.04	[12]	1.20	[13]	1.17
				[7]	3.98	[7]	2.04	[14]	1.19	[15]	1.21
						[8]	2.07	[16]	1.25	[17]	1.24
						[9]	2.01	[18]	1.19	[19]	1.23
						[10]	2.15	[20]	1.14	[21]	1.19
						[11]	2.10	[22]	1.19	[23]	1.17
						[12]	2.11	[24]	1.19	[25]	1.15
						[13]	2.10	[26]	1.14	[27]	1.11
						[14]	2.10	[28]	1.16	[29]	1.08
						[15]	2.08	[30]	1.11	[31]	1.13

Tabla 3.20: Tiempo de ejecución (en segundos) - 200,000 Observaciones - 16 Variables

200,000 Observaciones - 16 Variables												
Cantidad de nodos												
2		4		8		16		32				
[0]	4,803.90	[0]	2,420.56	[0]	1,220.30	[0]	671.67	[0]	463.00	[1]	465.18
[1]	4,849.52	[1]	2,427.37	[1]	1,214.41	[1]	669.34	[2]	468.06	[3]	468.18
		[2]	2,441.91	[2]	1,224.23	[2]	672.15	[4]	465.06	[5]	462.39
		[3]	2,432.42	[3]	1,223.44	[3]	664.01	[6]	461.51	[7]	464.11
				[4]	1,214.30	[4]	670.66	[8]	463.77	[9]	461.00
				[5]	1,219.91	[5]	668.63	[10]	466.07	[11]	464.78
				[6]	1,218.41	[6]	670.90	[12]	463.68	[13]	461.04
				[7]	1,216.37	[7]	665.71	[14]	462.25	[15]	462.56
						[8]	672.57	[16]	463.68	[17]	462.62
						[9]	666.24	[18]	466.18	[19]	465.60
						[10]	668.65	[20]	464.80	[21]	463.27
						[11]	660.97	[22]	460.30	[23]	462.23
						[12]	667.46	[24]	463.65	[25]	463.97
						[13]	665.02	[26]	464.98	[27]	465.00
						[14]	669.56	[28]	465.16	[29]	462.30
						[15]	662.43	[30]	462.14	[31]	463.20

Tabla 3.21: Tiempo de ejecución (en segundos) - 200,000 Observaciones - 20 Variables

200,000 Observaciones - 20 Variables												
Cantidad de nodos												
2		4		8		16		32				
[0]	90,913.84	[0]	45,487.27	[0]	22,863.68	[0]	13,291.17	[0]	10,073.24	[1]	10,034.34
[1]	91,409.61	[1]	45,860.38	[1]	22,924.97	[1]	13,276.33	[2]	10,094.39	[3]	10,057.51
		[2]	45,946.52	[2]	22,928.37	[2]	13,314.57	[4]	10,037.66	[5]	10,011.34
		[3]	45,519.18	[3]	22,848.92	[3]	13,160.47	[6]	10,014.70	[7]	10,022.65
				[4]	22,773.75	[4]	13,113.27	[8]	9,976.89	[9]	9,982.98
				[5]	22,861.75	[5]	13,115.01	[10]	10,028.35	[11]	10,026.16
				[6]	22,755.89	[6]	13,127.78	[12]	10,021.21	[13]	9,979.50
				[7]	22,791.30	[7]	13,132.17	[14]	9,978.37	[15]	9,990.49
						[8]	13,286.69	[16]	9,960.13	[17]	9,982.21
						[9]	13,147.99	[18]	10,022.24	[19]	10,023.35
						[10]	13,176.03	[20]	10,003.98	[21]	9,975.21
						[11]	13,120.68	[22]	9,966.41	[23]	9,988.99
						[12]	13,094.13	[24]	9,965.23	[25]	9,986.50
						[13]	13,068.48	[26]	10,026.46	[27]	10,016.37
						[14]	13,115.46	[28]	9,999.78	[29]	9,965.57
						[15]	13,101.06	[30]	9,966.35	[31]	9,978.65

Observando los tiempos de ejecución de los experimentos, se puede claramente distinguir que las diferencias son mucho menores de lo que eran con la asignación de tareas original. Usando el mismo ejemplo anterior, el de procesar 200,000 observaciones y 20 variables, utilizando 32 procesos, el menor tiempo de ejecución lo obtuvo el nodo 16 con 9,960.13 segundos, y el mayor tiempo lo obtuvo el nodo 2 con 10,094.39 segundos. Esto significa que la mayor diferencia es de 134.26 segundos, una mejora significativa con respecto al ejemplo anterior, en el cual la mayor diferencia fue de 3,984.67 segundos. Por lo tanto, se consiguió exitosamente tener un mejor balance de carga entre los procesos.

Conclusiones

En el presente trabajo de titulación se propusieron tres algoritmos para resolver el problema de la asignación de tareas en sistemas distribuidos. Un algoritmo implementa la técnica de *backtracking*, y los otros dos son soluciones nuevas, propuestas por el autor. En los experimentos, se concluyó que el algoritmo de Zigzag fue el que menos tiempo de ejecución tuvo, además de producir la matriz más deseable. Se aplicó el algoritmo de Zigzag para mejorar la asignación de tareas en el algoritmo distribuido para la prueba de normalidad exhaustiva, lo que ayudó a reducir de forma considerable el tiempo de ejecución total del algoritmo.

Como se señaló en el capítulo 3, en los experimentos, los tres algoritmos propuestos mostraron tiempos de ejecución mucho menores de los que se esperarían considerando sus complejidades de tiempo, por lo que un trabajo a futuro podría ser el de demostrar por qué ocurre esto.

Otro trabajo a futuro podría ser el de adaptar los algoritmos propuestos para que funcionen mejor en un sistema distribuido heterogéneo, esto es, un sistema distribuido en el cual los nodos que lo conforman no poseen características similares, lo que significa que algunos nodos tienen mayor poder computacional que otros nodos.

Por último, se podría aplicar el algoritmo de Zigzag en otro algoritmo distribuido para observar cuánto se reduce el tiempo de ejecución total en el sistema distribuido.

Bibliografía

Boakye, F., y Yao, S. (2016). Assessing univariate and multivariate normality, a guide for non statisticians. *Mathematical Theory and Modeling*, *6*.

Carvajal, R., y Moreno, F. (2022). *A distributed algorithm for exhaustive normality test.* (24th International Conference on Computational Statistics (COMPSTAT))

Chen, D.-S., Batson, R., y Dang, Y. (2010). *Applied integer programming* (1.ª ed.). John Wiley & Sons, Inc.

Erickson, J. (2019). *Algorithms* (1.ª ed.). Descargado 2023-03-20, de `https://jeffe.cs.illinois.edu/teaching/algorithms`

Knuth, D. (1984). Literate programming. *The Computer Journal*, *27*, 97-111.

Korkmaz, S., Goksuluk, D., y Zararsiz, G. (2014). Mvn: An r package for assessing multivariate normality. *The R Journal*, *6*.

Levitin, A. (2012). *Introduction to the design and analysis of algorithms* (3.ª ed.). Pearson Education.

Apéndice A

Códigos

En el código A.1 se muestra la implementación de los tres algoritmos propuestos, escritos en lenguaje C. Además, en el código A.2 se muestra el *script* escrito para Bash usado para realizar los experimentos.

Código fuente A.1: Implementación en C de los 3 algoritmos propuestos

```c
/*****************************************************************
 *
 * alloc.c
 *
 * Programmer: Claudio Saji Santander
 * Compile: gcc -o alloc.exe alloc.c -lm
 * Execute: ./alloc.exe m (exponent of 2) p (number of processes)
 *          -{zig, col, back} (optimization method)
 *          opt-matrix-m-p-{zig, col, back}.txt (optimized matrix in the txt file)
 *          exp-times-{zig, col, back}-p.txt (txt file for the experiments which contains
      the value of m and the time it took)
 *
 * Example: ./alloc.exe 8 4 -zig opt-matrix-8-4-zig.txt exp-times-zig-4.txt
 *
 *****************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <stdbool.h>
#include <sys/time.h>


```

```
25 struct ones {    // struct that enables to save the index of each sum of the "elements"
       matrix
26    unsigned int num;  // the number of active 1s in the "elements" matrix
27    unsigned int index;  // the index of the row in the "elements" matrix
28 };
29
30
31 struct matrix_size {   // struct that enables to return 2 values in a function, the
       matrix and its real size
32    struct ones **matrix;
33    unsigned int size;
34 };
35
36
37 void merge(struct ones arr[], int l, int m, int r) {
38     int i, j, k;
39     int n1 = m - l + 1;
40     int n2 = r - m;
41     struct ones L[n1], R[n2];
42
43     for (i = 0; i < n1; i++) {
44         L[i].num = arr[l + i].num;
45         L[i].index = arr[l + i].index;
46     }
47     for (j = 0; j < n2; j++) {
48         R[j].num = arr[m + 1 + j].num;
49         R[j].index = arr[m + 1 + j].index;
50     }
51
52     i = 0;
53     j = 0;
54     k = l;
55     while (i < n1 && j < n2) {
56         if (L[i].num <= R[j].num) {
57             arr[k].num = L[i].num;
58             arr[k].index = L[i].index;
59             i++;
60         }
61         else {
62             arr[k].num = R[j].num;
63             arr[k].index = R[j].index;
64             j++;
65         }
66         k++;
67     }
68
```

```
    while (i < n1) {
        arr[k].num = L[i].num;
        arr[k].index = L[i].index;
        i++;
        k++;
    }

    while (j < n2) {
        arr[k].num = R[j].num;
        arr[k].index = R[j].index;
        j++;
        k++;
    }
}


/*
 *
 */
void mergeSort(struct ones arr[], int l, int r) {
   int m;
   if (l < r) {
      m = l + (r - l) / 2;
      mergeSort(arr, l, m);
      mergeSort(arr, m + 1, r);
      merge(arr, l, m, r);
   }
}


/*
 *
 */
void PrintMatrix (struct ones **matrix, unsigned int f, unsigned int c) {
    unsigned int i, j;

    printf("\n\nMatrix=\n");
    for(i=0; i<f; ++i) {
        for(j=0; j<c; ++j)
            printf("%d,%d ", matrix[i][j].num, matrix[i][j].index);
        printf("\n");
    }
    printf("\n");
}


```

```
115 /*
116  *
117  */
118 void FilePrintMatrix (FILE *fp_matrix, struct ones **matrix, unsigned int f, unsigned int
        c) { // prints the matrix in the file
119     unsigned int i, j;
120
121     fprintf(fp_matrix, "%d %d\n", f, c);  // print the sizes
122     for(i=0; i<f; ++i) {
123         for(j=0; j<c; ++j)
124             fprintf(fp_matrix, "%d,%d ", matrix[i][j].num, matrix[i][j].index);
125         fprintf(fp_matrix, "\n");
126     }
127 }
128
129
130 /*
131  *
132  */
133 void CountNumbers (struct ones *arr, unsigned int size) {     // counts the number of
        times a number appears
134     unsigned int sum, i;
135     sum = 1;
136     for(i=0; i<size; ++i) {
137         if (i == size-1)
138             printf("%d -> %d times\n", arr[i].num, sum);
139         else if (arr[i].num == arr[i+1].num)
140             sum++;
141         else {
142             printf("%d -> %d times\n", arr[i].num, sum);
143             sum = 1;
144         }
145     }
146 }
147
148
149 /*
150  *
151  */
152 struct matrix_size ColFirst (struct ones *arr, unsigned int size, unsigned int c) {  //
        Columns first method
153     unsigned int i, j, total, ideal, f_ideal, sum, *col_sizes, max_row, max_num, min_num,
            r, init;
154     int k, rem, q, min_q, z;
155     struct ones **matrix;
156     struct matrix_size output_matrix;   // the matrix to return with its real size (it's
```

```
        used to be able to return 2 values in this function)
157     bool rdy, key, all_done;
158
159     r = size/c;
160     matrix = (struct ones **) malloc(sizeof(struct ones*) * r);
161     for(i=0; i<r; ++i)
162         matrix[i] = (struct ones *) malloc(sizeof(struct ones) * c);
163
164     col_sizes = malloc(sizeof(unsigned int) * c);
165
166     total = 0;
167     for(i=0; i<size; ++i)
168         total = total + arr[i].num;   // the total sum of all the numbers in the array
169
170     ideal = total/c;   // the ideal sum to have in all the columns
171     f_ideal = ideal;   // fixed ideal that is used for when there is no remainder left
172     rem = total%c;     // the total number of columns that will have (f_ideal + 1) as sum
173     if (rem > 0)
174         ideal++;
175     printf("\nTotal = %d\nIdeal = %d\nRemainder = %d\n\n", total, f_ideal, rem);
176
177     max_row = 0;
178     rdy = false;  // flag used for when a column is ready. Skip the rest of the column
        and go to the next one
179     all_done = false;  // flag that is used for breaking the principal loop when all the
        numbers of the array have been assigned in the matrix
180     sum = 0;
181     k = size - 1;   // k is the latest index that has a value of arr[k].num != 0.
182     max_num = arr[k].num;
183     for(j=0; j<c; ++j) {   // principal 'for' that traverses all the matrix. It breaks
        when all the numbers in the array have been assigned in the matrix
184         for(i=0; i<size; ++i) {
185             if (i == r) {
186                 matrix = (struct ones **) realloc(matrix, sizeof(struct ones*) * (r+=256)
                    );  //add space for another 256 rows
187                 for(init=i; init<r; ++init)
188                     matrix[init] = (struct ones *) malloc(sizeof(struct ones) * c);
189             }
190
191             if (arr[k].num == 0) {    // k has to be the latest index that has a value of
                  arr[k].num != 0. The numbers are marked as 0 when they have been used
192                 all_done = true;  // for cases where all the numbers to the left of k
                    have been used, and also k has been used
193                 for(z = k - 1; z >= 0; --z) {
194                     if (arr[z].num != 0) {
195                         k = z;
```

```
196                     all_done = false;
197                     break;
198                 }
199             }
200             if (all_done == true)
201                 break;
202         }
203 
204         if (rem <= 0)  // if there is no remainder left the value of ideal decreases
                by 1
205             ideal = f_ideal;
206 
207         if (sum + arr[k].num <= ideal) {   // the number can be assigned
208             if (sum + arr[k].num == ideal) {   // the sum is exactly the same to the
                    ideal sum
209                 if (rem > 0)
210                     rem--;
211                 rdy = true;  // the column is ready so set the flag as active
212             }
213             matrix[i][j].num = arr[k].num;
214             matrix[i][j].index = arr[k].index;
215             sum = sum + matrix[i][j].num;
216             arr[k].num = 0;  // mark the number as used
217             if (k > 0)
218                 k--;
219             if (rdy == true && j != c-1) {   // if the column is ready and is not the
                     last one, skip to the next column
220                 rdy = false;   // set the flag as inactive for the next column
221                 break;  // skip to the next column
222             }
223         }
224 
225         else {   // search for a smaller number that can be assigned
226             key = false;   // a flag that is used for when no smaller number was
                    found
227             min_num = max_num;  // the minimum number found in the search needs to be
                     set in the beginning as the maximum number in the array (for
                    comparison)
228             for(q = k; q >= 0; --q) {
229                 if (arr[q].num != 0 && arr[q].num < min_num) {  // a new minimum
                        number was found
230                     min_num = arr[q].num;
231                     min_q = q;  // save the index in the array of the minimum number
                            found
232                 }
233                 if (arr[q].num != 0 && sum + arr[q].num <= ideal) {   // a smaller
```

```
                    number was found
234                 if (sum + arr[q].num == ideal) {  // adding the smaller number
                        leaves the sum with the same value of "ideal"
235                     if (rem > 0)
236                         rem--;
237                     rdy = true;   // the column is ready so mark the flag as
                            active
238                 }
239                 matrix[i][j].num = arr[q].num;
240                 matrix[i][j].index = arr[q].index;
241                 sum = sum + matrix[i][j].num;
242                 arr[q].num = 0;   // mark the number arr[q].num as used
243                 key = true;   // mark the flag as active indicating that a number
                         was found
244                 break;
245             }
246         }
247         if (rdy == true && j != c-1) {   // if the column is ready and is not the
                 last one, skip to the next column
248             rdy = false;   // set the flag as inactive for the next column
249             break;  // skip to the next column
250         }
251         if (key == false) {   // cases where no smaller number that can be
                assigned was found
252             matrix[i][j].num = arr[min_q].num;
253             matrix[i][j].index = arr[min_q].index;
254             sum = sum + matrix[i][j].num;
255             if (rem > 0)
256                 rem = rem - 1 - (sum - ideal);  // decrease the value of the
                        remainder according to the difference of the ideal sum and
                        actual sum
257             arr[min_q].num = 0;   // mark the number arr[min_q].num as used
258             if (j != c-1)  // only if this column is not the last one, skip to
                    the next one
259                 break;  // skip to the next column
260         }
261     }
262 }
263 if (j == c-1)
264     col_sizes[j] = i;   // save the size of the last column
265 else
266     col_sizes[j] = i+1;   // save the size of the column
267 if (col_sizes[j] > max_row)
268     max_row = col_sizes[j];  // save the size of the row which has the greatest
            size. It's used later to know the real size of the matrix
269 if (all_done == true)  // break the principal loop when all the numbers have been
```

```
             assigned
270             break;
271         sum = 0;   // reset the sum for the next column
272     }
273
274     for(j=0; j<c; ++j) {   // fill the rest of the matrix with zeros up to "max_row"
275         for(i=col_sizes[j]; i<max_row; ++i) {
276             matrix[i][j].num = 0;
277             matrix[i][j].index = 0;
278         }
279     }
280
281     matrix = (struct ones **) realloc(matrix, sizeof(struct ones*) * max_row);  // cut
        the unused part of the matrix
282     output_matrix.matrix = matrix;
283     output_matrix.size = max_row;
284     return output_matrix;  // return the struct that has the matrix and its real size
285 }
286
287
288 /*
289  *
290  */
291 void AssignNumber (struct ones **matrix, struct ones *arr, unsigned int *sums, unsigned
    int i, unsigned int *ideal, unsigned int f_ideal,
292                    int j, int *k, int *rem, bool *rdy, unsigned int max_num) {   //
                       assign a number from the array in the matrix
293     int q, z, min_q;  // ideal, k, rem are "passed by reference"
294     unsigned int min_num;
295     bool key;
296
297     if (*rem <= 0)
298         *ideal = f_ideal;
299
300     if (*k == -1) {   // there are no numbers left in the array
301         matrix[i][j].num = 0;
302         matrix[i][j].index = 0;
303         return;
304     }
305     if (arr[*k].num == 0) {      // k has to be the latest index that has a value of arr[
        k].num != 0
306         for(z = *k - 1; z >= 0; --z) {
307             if (arr[z].num != 0) {
308                 *k = z;
309                 break;
310             }
```

```
311         }
312     }
313     if (rdy[j] == true) {   // first case: the column 'j' is ready
314         matrix[i][j].num = 0;
315         matrix[i][j].index = 0;
316     }
317     else if (sums[j] == *ideal) {   // second case: if (rem <= 0 && sums[j] == f_ideal)
318         rdy[j] = true;              // happens when all the columns with the remainder are
                ready and the sum of the column 'j' was not marked as ready
319         matrix[i][j].num = 0;
320         matrix[i][j].index = 0;
321     }
322     else if (sums[j] + arr[*k].num <= *ideal) {   // third case: the number arr[k].num
            can be assign to this cell
323         if (sums[j] + arr[*k].num == *ideal) {   // the sum is exactly the same to "ideal
                "
324             rdy[j] = true;   // mark the column 'j' as ready
325             if (*rem > 0)
326                 *rem = *rem - 1;
327         }
328         matrix[i][j].num = arr[*k].num;
329         matrix[i][j].index = arr[*k].index;
330         sums[j] = sums[j] + matrix[i][j].num;
331         *k = *k - 1;
332     }
333     else {   // fourth case: the number arr[k].num cannot be assign to this cell, so
            search for a smaller number
334         key = false;   // a flag that is used for when no smaller number was found
335         min_num = max_num;  // the minimum number found in the search needs to be set in
                the beginning as the maximum number in the array (for comparison)
336         for(q = *k - 1; q >= 0; --q) {
337             if (arr[q].num != 0 && arr[q].num < min_num) {  // a new minimum number was
                    found
338                 min_num = arr[q].num;
339                 min_q = q;  // save the index in the array of the minimum number found
340             }
341             if (arr[q].num != 0 && sums[j] + arr[q].num <= *ideal) {   // a smaller
                    number was found
342                 if (sums[j] + arr[q].num == *ideal) {  // the sum is exactly the same to
                        "ideal"
343                     rdy[j] = true;   // mark the column 'j' as ready
344                     if (*rem > 0)
345                         *rem = *rem - 1;
346                 }
347                 matrix[i][j].num = arr[q].num;
348                 matrix[i][j].index = arr[q].index;
```

```
349                 sums[j] = sums[j] + matrix[i][j].num;
350                 arr[q].num = 0;   // mark the number arr[q].num as used
351                 key = true;   // mark the flag as active indicating that a number was
                        found
352                 break;
353             }
354         }
355         if (key == false) {   // if a smaller number was not found, assign the minimum
                number found
356             rdy[j] = true;  // mark the column j as ready because it exceeds the ideal
                    sum
357             matrix[i][j].num = arr[min_q].num;
358             matrix[i][j].index = arr[min_q].index;
359             sums[j] = sums[j] + matrix[i][j].num;
360             if (*rem > 0)
361                 *rem = *rem - 1 - (sums[j] - *ideal);  // decrease the value of the
                        remainder according to the difference of the ideal sum and actual sum
362             arr[min_q].num = 0;   // mark the number arr[min_q].num as used
363         }
364     }
365 }
366
367
368 /*
369  *
370  */
371 struct matrix_size ZigzagCheck (struct ones *arr, unsigned int size, unsigned int c) {
        // Zigzag method
372     unsigned int *sums, i, total, ideal, f_ideal, real_size, max_num, r, init;
373     int j, k, rem;
374     bool *rdy, dir;
375     struct ones **matrix;   // the matrix that will have all its columns with similar sum
376     struct matrix_size output_matrix;   // the matrix to return with its real size (it's
            used to be able to return 2 values in this function)
377
378     rdy = malloc(c * sizeof(bool));   // Boolean array that keeps track on the columns
            that are ready (it's necesary because the value of "ideal" changes if "rem"
            changes)
379     for(i=0; i<c; ++i)
380         rdy[i] = false;
381
382     r = size/c;  // give an initial number of rows that later will be cut or extended
            with realloc
383     matrix = (struct ones **) malloc(sizeof(struct ones*) * r);
384     for(i=0; i<r; ++i)
385         matrix[i] = (struct ones *) malloc(sizeof(struct ones) * c);
```

```
386
387     sums = malloc(c * sizeof(unsigned int));   // array with the temporal sums of the
            columns
388     for(i=0; i<c; ++i)
389         sums[i] = 0;
390
391     total = 0;
392     for(i=0; i<size; ++i)
393         total = total + arr[i].num;   // the total sum of all the numbers in the array
394
395     ideal = total/c;   // the ideal sum to have in all the columns
396     f_ideal = ideal;   // fixed ideal that is used for when there is no remainder left
397     rem = total%c;     // the total number of columns that will have (f_ideal + 1) as sum
398     if (rem > 0)
399         ideal++;
400     printf("\nTotal = %d\nIdeal = %d\nRemainder = %d\n\n", total, f_ideal, rem);
401
402     k = size - 1;   // k is the latest index that has a value of arr[k].num != 0. The
            array needs to be sorted by lowest to highest
403     max_num = arr[k].num;
404     dir = false;   // direction flag used for changing direction in the traversal of the
            rows. First, from left to right, then right to left, then left to right again.
            Like a zigzag
405     for(i=0; i<size; ++i) {   // principal 'for' that traverses all the matrix. It breaks
             when all the numbers in the array have been assigned in the matrix
406         if (i == r) {
407             matrix = (struct ones **) realloc(matrix, sizeof(struct ones*) * (r+=256));
                    //add space for another 256 rows
408             for(init=i; init<r; ++init)
409                 matrix[init] = (struct ones *) malloc(sizeof(struct ones) * c);
410         }
411         if (dir == false) {
412             for(j=0; j<c; ++j) {   // traverse a row from left to right
413                 AssignNumber(matrix, arr, sums, i, &ideal, f_ideal, j, &k, &rem, rdy,
                        max_num);
414             }
415             if (k == -1) {     // all the numbers in the array have been assigned in the
                    matrix
416                 real_size = i+1;  // the real size of the matrix is used later for the
                        realloc
417                 break;   // break the principal 'for'
418             }
419             dir = true;  // change the flag so the direction of the traversal changes
420         }
421         else {
422             for(j=c-1; j>=0; --j) {   // traverse a row from right to left
```

```
423                 AssignNumber(matrix, arr, sums, i, &ideal, f_ideal, j, &k, &rem, rdy,
                        max_num);
424             }
425             if (k == -1) {
426                 real_size = i+1;
427                 break;
428             }
429             dir = false;
430         }
431     }
432
433     matrix = (struct ones **) realloc(matrix, sizeof(struct ones*) * real_size);  // cut
            the unused part of the matrix
434     output_matrix.matrix = matrix;
435     output_matrix.size = real_size;
436     return output_matrix;  // return the struct that has the matrix and its real size
437 }
438
439
440 /*
441  *
442  */
443 bool backtrack (struct ones *arr, struct ones **matrix, unsigned int *sums, unsigned int
        size, unsigned int c,
444                 unsigned int ideal, unsigned int rem, unsigned int f_ideal, unsigned int
                        x, unsigned int y,
445                 bool *used, bool *rdy, unsigned int all_rdy, unsigned int *real_size) {
                        // permutes the numbers until it finds a solution
446     int i, j, rp;

            // begins from the end of the array
447     unsigned int init;
448     rp = -1;
449     if (all_rdy == c) {  // Solution found, all the columns have the same sum
450         for(j=y; j<c; ++j) {  // fill the rest of the row with zeros
451             matrix[x][j].num = 0;
452             matrix[x][j].index = 0;
453         }
454         *real_size = x+1;  // assign the real size of the matrix to cut the rest with
                realloc later
455         return true;
456     }
457     else {
458         for(i=size-1; i>=0; --i) {  // begin from the end of the array
459             if (rp == arr[i].num) continue;  // skips the repeated numbers that are known
                     to cause a backtrack
```

```
460
461             if (rem == 0)
462                 ideal = f_ideal;
463             else
464                 ideal = f_ideal + 1;
465
466             if (rdy[y] == true) {  // first case: the column 'y' is ready. Assign a 0 to
                    the cell
467                 matrix[x][y].num = 0;
468                 matrix[x][y].index = 0;
469
470                 if (y == c-1) {   // if this cell is in the last column, call the
                        recursive function to the next row (x+1) and next column (0)
471                     if (backtrack(arr, matrix, sums, size, c, ideal, rem, f_ideal, x+1,
                            0, used, rdy, all_rdy, real_size) == true)
472                         return true;  // if the recursive function returns true, this
                                instance returns true also
473                 }                     // this is used for finding just one solution
474                 else {   // if this cell is not in the last column, call the recursive
                        function to the same row (x) and next column (y+1)
475                     if (backtrack(arr, matrix, sums, size, c, ideal, rem, f_ideal, x, y
                            +1, used, rdy, all_rdy, real_size) == true)
476                         return true;
477                 }
478                 return false;  // if the recursive function returns false, there is no
                        other option but to return false also
479             }
480
481             else if (rem == 0 && sums[y] == f_ideal) {  // second case: happens when all
                    the columns with the remainder are ready and
482                 matrix[x][y].num = 0;                   //              the sum of the
                        column 'y' was not marked as ready.
483                 matrix[x][y].index = 0;                 // it happens because the value
                        of "ideal" changes based in the remainder
484                 rdy[y] = true;
485                 all_rdy++;
486
487                 if (y == c-1) {   // last column
488                     if (backtrack(arr, matrix, sums, size, c, ideal, rem, f_ideal, x+1,
                            0, used, rdy, all_rdy, real_size) == true)
489                         return true;
490                 }
491                 else {
492                     if (backtrack(arr, matrix, sums, size, c, ideal, rem, f_ideal, x, y
                            +1, used, rdy, all_rdy, real_size) == true)
493                         return true;
```

```
494                 }
495                 rdy[y] = false;  // the "remove" part of the backtrack algorithm. Remove
                        the changes done before
496                 all_rdy--;
497                 return false;  // no other option
498             }
499             else if (used[i] == false) {   // third case: the column 'y' is not ready, so
                     search for a number to assign.
500                 if (sums[y] + arr[i].num <= ideal) {  // check if the number arr[i] is
                        not in the matrix yet with the array "used"
501                     if (sums[y] + arr[i].num == ideal) {  // the sum is exactly the same
                            to "ideal"
502                         if (rem > 0)
503                             rem--;
504                         rdy[y] = true;  // mark the column 'y' as ready
505                         all_rdy++;   // add one to the counter of columns that are ready
506                     }
507                     used[i] = true;   // mark the number arr[i] as used, so it is not
                            used again
508                     matrix[x][y].num = arr[i].num;   // assign the number to the matrix
509                     matrix[x][y].index = arr[i].index;   // assign the index to the
                            matrix
510                     sums[y] = sums[y] + arr[i].num;   // update the array with the
                            temporary sums of the columns
511
512                     if (y == c-1) {   // last column
513                         if (backtrack(arr, matrix, sums, size, c, ideal, rem, f_ideal, x
                                +1, 0, used, rdy, all_rdy, real_size) == true)
514                             return true;
515                     }
516                     else {
517                         if (backtrack(arr, matrix, sums, size, c, ideal, rem, f_ideal, x,
                                 y+1, used, rdy, all_rdy, real_size) == true)
518                             return true;
519                     }
520
521                     printf("\n----REMOVE PART-----\nrem=%d, all_ready=%d", rem, all_rdy);
522                     // "remove part" of the backtrack algorithm
523                     rp = arr[i].num;  // save the number arr[i] so it is skipped in the
                            next iteration, because now it is known to cause a backtrack
524
525                     if (sums[y] == f_ideal + 1) // a column that was ready with the
                            remainder included
526                         rem++;  // restore the value of the remainder
527                     if (rdy[y] == true) {  // if the column was ready, now its not going
                            to be, because a number distinct from 0 was removed from the
```

```
                        column
528                     all_rdy--;  // update the value of the counter
529                     rdy[y] = false;
530                 }
531                 used[i] = false;
532                 matrix[x][y].num = 0;
533                 matrix[x][y].index = 0;
534                 sums[y] = sums[y] - arr[i].num;
535             }
536         }
537     }
538     return false;  // end of the "for" loop: no number that can be assign was found,
            so return false and backtrack
539 }
540 }
541
542
543 /*
544  *
545  */
546 struct matrix_size permute (struct ones *arr, unsigned int size, unsigned int c) {  //
        Backtracking method
547     unsigned int *sums, i, j, total, ideal, rem, f_ideal, real_size;
548     bool *used, *rdy;
549     struct ones **matrix;   // the matrix that will have all its columns with similar sum
550     struct matrix_size output_matrix;   // the matrix to return with its real size (it's
            used to be able to return 2 values in this function)
551
552     rdy = malloc(c * sizeof(bool));  // boolean array that keeps track of the columns
            that are ready (it's necessary because the value of "ideal" changes)
553     for(i=0; i<c; ++i)
554       rdy[i] = false;
555
556     used = malloc(size * sizeof(bool));  // boolean array that marks the numbers in "arr"
             as used or not
557     for(i=0; i<size; ++i)
558       used[i] = false;
559
560     matrix = (struct ones **) malloc(sizeof(struct ones*) * size);
561     for(i=0; i<size; ++i)
562       matrix[i] = (struct ones *) malloc(sizeof(struct ones) * c);
563
564     sums = malloc(c * sizeof(unsigned int));   // array with the temporal sums of the
            columns
565     for(i=0; i<c; ++i)
566       sums[i] = 0;
```

```
567
568     total = 0;
569     for(i=0; i<size; ++i)
570        total = total + arr[i].num;
571
572     ideal = total/c;
573     f_ideal = ideal;
574     rem = total%c;
575     if (rem > 0)
576         ideal++;
577
578     printf("\nTotal = %d\nIdeal = %d\nRemainder = %d\n\n", total, f_ideal, rem);
579     backtrack(arr, matrix, sums, size, c, ideal, rem, f_ideal, 0, 0, used, rdy, 0, &
            real_size);
580     matrix = (struct ones **) realloc(matrix, sizeof(struct ones*) * real_size);  // cut
            the unused part of the matrix
581     output_matrix.matrix = matrix;
582     output_matrix.size = real_size;
583     return output_matrix;   // return the matrix and its real size
584 }
585
586
587 /*
588  *
589  */
590 void SumCol (struct ones **matrix, unsigned int f, unsigned int c) {   // sums the
        columns of the matrix to check that all have similar sum (for testing)
591    unsigned int i, j, sum, total;
592
593    total = 0;
594    for(j=0; j<c; ++j) {
595       sum = 0;
596       for(i=0; i<f; ++i)
597         sum = sum + matrix[i][j].num;
598       printf("%d, ", sum);
599       total = total + sum;
600    }
601    printf("\nIdeal = %d\nRemainder = %d\n", total/c, total%c);
602 }
603
604
605 /*
606  *
607  */
608 void main(unsigned int argc, char **argv) {
609     unsigned int n, m, i, j, k, mask, sum, total, p, t_size;
```

```
610     struct matrix_size output_matrix;
611     struct ones **matrix_ones, *arr_ones;
612     struct timeval t0, t1, dt;
613     char opt_matrix_txt[30], exp_times_txt[25];
614     FILE *fp_time, *fp_opt_matrix;
615
616     m = atoi(argv[1]);
617     p = atoi(argv[2]);
618     total = pow(2,m) - 1;
619     arr_ones = (struct ones *) malloc(sizeof(struct ones) * total);
620
621     for (i=1; i <= total; ++i) {
622         sum = 0;
623       mask = 1;
624       for (j=0, k=1; j < m; ++j) {
625           mask = 1 << j;
626           if ((mask & i) != 0)
627               sum++;
628       }
629       arr_ones[i-1].num = sum;  // assign the number of active ones and the index of the
              row in the "elements" matrix to not lose it later
630         arr_ones[i-1].index = i;
631     }
632
633     //mergeSort(arr_ones, 0, total - 1);  // optional
634
635     /*
636     printf("\nOnes=\n");
637     for(i=0; i<total; ++i)
638         printf("%d,%d\n", arr_ones[i].num, arr_ones[i].index);
639     */
640
641     /*
642     printf("\n\n\nQuantity = \n");
643     CountNumbers(arr_ones, total);
644     */
645     strcpy(exp_times_txt, argv[5]);  // copy the name of the txt file into the buffer
646     gettimeofday(&t0,NULL);
647
648     if (total <= p) {   // no need to create a matrix and use any optimization method
649         strcpy(opt_matrix_txt, argv[4]);  // copy the name of the txt file into the
              buffer
650         fp_opt_matrix = fopen(opt_matrix_txt, "w");
651         fprintf(fp_opt_matrix, "1 %d\n", p);  // print the sizes
652         for (i=0; i<p; ++i) {
653             if (i < total)
```

```
654                 fprintf(fp_opt_matrix, "%d,%d ", arr_ones[i].num, arr_ones[i].index);
655             else
656                 fprintf(fp_opt_matrix, "0,0 ");
657         }
658 
659         fclose(fp_opt_matrix);
660         gettimeofday(&t1,NULL);
661         timersub(&t1,&t0,&dt);
662         fp_time = fopen(exp_times_txt,"a");
663         fprintf(fp_time, "%d %ld.%06ld\n", m, dt.tv_sec, dt.tv_usec);  // print on the
                file: m (the exponent of 2) and the time it took
664         fclose(fp_time);
665         free(arr_ones);  // free the memory
666         arr_ones = NULL;
667         return;
668     }
669 
670     else if (strcmp(argv[3],"-col") == 0) {
671         output_matrix = ColFirst(arr_ones, total, p);
672         fp_time = fopen(exp_times_txt,"a"); // append the time on the experiment times
                file for the Columns First method
673     }
674 
675     else if (strcmp(argv[3],"-zig") == 0) {
676         output_matrix = ZigzagCheck(arr_ones, total, p);
677         fp_time = fopen(exp_times_txt,"a"); // append the time on the experiment times
                file for the Zigzag method
678     }
679 
680     else if (strcmp(argv[3],"-back") == 0) {
681         output_matrix = permute(arr_ones, total, p);
682         fp_time = fopen(exp_times_txt,"a"); // append the time on the experiment times
                file for the Backtracking method
683     }
684 
685     gettimeofday(&t1,NULL);
686     timersub(&t1,&t0,&dt);
687     //printf("\nElapsed Wall Time = %ld.%06ld Seconds \n",dt.tv_sec, dt.tv_usec);
688     //fflush(stdout);
689 
690     fprintf(fp_time, "%d %ld.%06ld\n", m, dt.tv_sec, dt.tv_usec);  // print on the file:
            m (the exponent of 2) and the time it took
691     fclose(fp_time);
692 
693     free(arr_ones);  // free the memory
694     arr_ones = NULL;
```

```
695
696     strcpy(opt_matrix_txt, argv[4]);  // copy the name of the txt file into the buffer
697     fp_opt_matrix = fopen(opt_matrix_txt, "w");
698     //PrintMatrix(output_matrix.matrix, output_matrix.size, p);
699     //SumCol(output_matrix.matrix, output_matrix.size, p);  // to check that the columns
            have similar sum (for testing)
700     FilePrintMatrix(fp_opt_matrix, output_matrix.matrix, output_matrix.size, p);  //
            print the matrix in the file
701     fclose(fp_opt_matrix);
702
703     for (i=0; i<output_matrix.size; ++i) { // free the memory
704         free(output_matrix.matrix[i]);
705         output_matrix.matrix[i] = NULL;
706     }
707     free(output_matrix.matrix);
708     output_matrix.matrix = NULL;
709 }
```

Código fuente A.2: *script* de Bash para los experimentos

```
1  #!/bin/bash
2  # method: optimization method
3  # m: variables
4  # p: number of processes
5
6  for method in zig col
7  do
8      for m in 4 8 16 20 24
9      do
10         for p in 2 4 8 16 32
11         do
12             ./alloc.exe $m $p -$method opt-matrix-$m-$p-$method.txt exp-times-$method-$p.
                   txt
13         done
14     done
15 done
16
17 # The backtracking method only supports for m <= 15
18
19 for m in 4 8 15
20 do
21     for p in 2 4 8 16 32
22     do
23         ./alloc.exe $m $p -back opt-matrix-$m-$p-back.txt exp-times-back-$p.txt
24     done
25 done
```