UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERIA
ESCUELA DE CIENCIAS Y SISTEMAS
LABORATORIO ESTRUCTURA DE DATOS

Manual Técnico Fase 3

Pixel Print Studio

Nombre: Sergio Joel Rodas Valdez
Carné: 202200271
Sección: C

Introducción

Este manual técnico, desarrollado como parte esencial del proyecto Pixel Print Studio, provee una visión detallada de la infraestructura lógica de la aplicación. A lo largo de este documento, se desglosan los componentes fundamentales que forman la base de la aplicación, revelando cómo las estructuras tanto lineales como no lineales desempeñan roles cruciales en la organización y manejo de datos, La descripción de estas estructuras, junto con la explicación de las técnicas Más allá de la simple gestión de datos, este manual destaca la implementación de algoritmos avanzados para la optimización de rutas, esenciales para la operativa diaria del negocio.

La aplicación no solo utiliza el algoritmo de costo uniforme para determinar la ruta más eficiente en términos de distancia, sino que también explora métodos como la búsqueda por anchura para maximizar la efectividad operacional en términos de recursos y tiempo, adaptándose a diversas necesidades.

La integración de tecnologías como el blockchain y el árbol de Merkle es particularmente notable, ya que estas no solo incrementan la transparencia y la trazabilidad de las transacciones, sino que también fortalecen la integridad de los datos.

Manual Técnico

Este manual detalla la lógica subyacente de la aplicación como parte del proyecto continuo Pixel Print Studio. Se abordará tanto estructuras lineales como no lineales y se explicará el proceso de escritura de un archivo JSON para asegurar la persistencia de los datos. El manual se organizará en varias secciones, cada una dedicada a explicar en profundidad el contenido específico de los distintos componentes.

- Inicio de Sesión

  Para garantizar la seguridad y la integridad de los datos, el acceso al sistema está restringido a empleados autorizados es decir solo al administrado. El usuario admin debe ingresar al sistema utilizando su identificación única y una contraseña personal. Estos datos deben ser ingresados en la pantalla de inicio de sesión que aparece al acceder al sistema.

Sucursales

La estructura de sucursales está compuesta por árbol bst este árbol este compuesto por nodos sucursales donde cada sucursal poseerá información valiosa para realizar el grafo, cada sucursal se debe registrar en el sistema con su información básica, que incluye: ID: Identificador único de la sucursal, contraseña: Contraseña para ingreso de acciones posteriormente se describirá estas acciones, departamento Ubicación geográfica, dirección: Dirección física completa, a continuación en la imagen interior de muestra la estructura del árbol.

```
type node_sucursal
    integer :: id_sucursal
    character(len=30) :: departament
    character(len=32) :: dirrecion
    character(len=32) :: password
    type(hash), allocatable :: tecnicos
    type(node_sucursal), pointer :: left => null()
    type(node_sucursal), pointer :: right => null()
end type node_sucursal

type :: bst
    type(node_sucursal), pointer :: root => null()
    contains
        procedure :: add_bst
        procedure :: search_node_bst
        procedure :: inorderTraversal
        procedure :: search_node_bst_id !buscar una sucursal por id
        procedure :: write_bst_to_json
end type
```

Rutas

Las rutas esta compuesto por una lista de adyacencia donde cada lista tiene vértices y aristas los vértices son las sucursales que están en el grafo y los vértices es la conexión que hay entre una sucursal a otra esto es muy importante en la aplicación ya que con esta información se realizara todas las rutas optimas esta parte viene cuando el administrador

carga el archivo json de las rutas es ahí donde parte la creación del grafo como tal para encontrar los posibles caminos, a continuación se muestra la estructura de la lista de adyacencia para el grafo.

```
!::::::: Estrcutura del grafo ( lista de adyacencia )
! arista
type arista
    integer :: distance  ! distancia entre nodo Origen - nodo Vecino
    integer :: print_mant ! impresoar a mantenimiento
    type(arista), pointer ::  next_ari  ! siguiente arista
    type(vertice), pointer :: dest_vertex ! vertice destino
    logical :: ari_visitada = .false.
end type arista

!Vertice == nodo
type vertice
    integer :: vertex_id
    type(vertice), pointer :: next_vertex
    type(arista), pointer :: ari

    integer :: acummulated_dis = 9999  ! acumular distancia
    integer :: printMant_dis = 0    ! acumular impresoras arregladas
    logical :: visited = .false.
    type(vertice), pointer :: parent_nodo  ! para almacenar el la ruta padre -> hijo
end type vertice
```

```
! ::: Estructura para aguardar mis rutas
type ruta
    type(nodo_vertice), pointer :: vertice_head ! vertice cabeza
    integer :: total_impresoras ! impresoras reparadas
    integer :: total_distancia ! total distancia
    type(ruta), pointer :: next_route ! siguiente ruta
end type ruta

type grafo
    type(vertice), pointer :: first_vertex
    integer :: zise_graph
    contains
        procedure :: initiGrafo
        procedure :: InsertArista
        procedure :: InsertaVertice
        procedure :: MostrarListaAdyacencia
        procedure :: GenerarGrafoDot
        procedure :: initialize_parameters !Inicializar parametros para reutilizar best_rooute
        procedure :: best_route ! Encontrar la distancia minima del grafo
        procedure :: best_route_maxPrintMant
        procedure :: find_all_routes
        procedure :: optimal_route
        procedure :: graph_best_route
end type grafo
```

Cálculo de Rutas Óptimas

En esta parte me quiero centrar mas en como fue elaborado el calculo de la ruta optimas esta parte es muy importante ya que en ella es como se centra las mejores rutas, el grafo es dirigido y cuenta con dos pesos el primero peso es la distancia y el segundo es la cantidad de impresoras a reparar para el caso de la distancia mínima se utilizo el costo uniforme, El algoritmo de costo uniforme, también conocido como búsqueda de costo

uniforme, es una estrategia Este algoritmo selecciona el camino más corto en un grafo ponderado sin tener en cuenta el objetivo final Es especialmente útil en situaciones donde el camino a seguir es menos obvio es por eso que se tomó este algoritmo para encontrar el camino con menor distancia la estructura utilizada es la siguiente la que se estableció en la aplicación.

Cola de Prioridad: Es la estructura de datos central en la implementación del algoritmo de costo uniforme. La cola de prioridad almacena todos los nodos que aún no se han explorado, ordenados por el costo del camino en este caso la prioridad es la menor distancia

El algoritmo se basa en la siguiente funcionalidad:
inicialización: Se empieza en el nodo raíz y se añade a la cola de prioridad.
Revisión de Nodos: Si un nodo adyacente ha sido visitado anteriormente con un costo mayor, se actualiza su costo en la cola de prioridad. Si el nodo no está en la cola, se añade con su nuevo costo. Es decir, busca el menor costo y conforme a la menos distancia así se ingresa a la cola de prioridad. El proceso se repite hasta que se alcanza el nodo objetivo o hasta que la cola de prioridad se vacíe, lo que indica que no hay más nodos accesibles.

```
    if(associated(current))then
        if(current%vertex_id == destino)then
            print *, "Rutas mas corta"
            call show_route(current) ! obtenendo el nodo padre de la ruta
            print *, ""
            print *, "Distancia del camino mas corto: ", current%acummulated_dis
            print *, "Impresoras reparadas: ", current%printMant_dis
            costos = (current%acummulated_dis)*80 !costo por la ruta minima
            ganancias = (current%printMant_dis)*100 !ganancias por la ruta minima
            print *, "Costos: ", costos
            print *, "Ingresos Extras: ", ganancias
            print *, ""
            !valores a retornar
            call get_route(current, ruta, count)
            total_distancia = current%acummulated_dis
            total_impresoras = current%printMant_dis
            completed_successfully = .true.
            return
        end if
    end if
    current%visited = .true.
    arista_actual => current%ari

    do while(associated(arista_actual))
        vecino => arista_actual%dest_vertex ! vertice venico o destino
        if(.not. vecino%visited)then
            if(vecino%acummulated_dis > current%acummulated_dis + arista_actual%distance) then
                vecino%acummulated_dis = current%acummulated_dis + arista_actual%distance
                vecino%printMant_dis = current%printMant_dis + arista_actual%print_mant
                vecino%parent_nodo => current
                call tail%insert_priority_queue(vecino) ! insertar a la cola de prioridad
            end if
        end if
        arista_actual => arista_actual%next_ari ! aqui
    end do
end do
```

Para encontrar el camino con mayor número de impresoras aquí se utilizo una búsqueda por anchura donde la intención es pretender buscar todas las posibles rutas que tiene el grafo para esta parte se realiza primero la búsqueda por anchura que después se aguardara en una lista para tener referencia de todos los posibles caminos que tiene el grafo después que se encontró todas las posibles rutas es aquí donde se comienza a organizador y extraer la primera ruta con mayor numero de impresoras a reparar como se muestra en la imagen esta subrutina es la encarga de realizar todo el proceso para buscar todos los caminos posibles.

```
    if (associated(current) .and. current%vertex_id == destination%vertex_id) then
        ! Crear una nueva ruta
        allocate(new_route)
        new_route%vertice_head => route_head
        new_route%total_distancia = total_dist
        new_route%total_impresoras = total_print_mant
        new_route%next_route => null()
        call addRoute(all_routes, new_route) ! añadir ruta
        is_route_added = .true.
    else
        edge => current%ari
        do while (associated(edge))
            neighbor => edge%dest_vertex
            is_associated = .false.
            ! Verificar si el vecino ya está en la ruta actual para evitar ciclos
            last_node => route_head
            do while (associated(last_node))
                if (last_node%vertice%vertex_id == neighbor%vertex_id) then
                    !ultimo%vertice%id_vertice
                    is_associated = .true.
                    exit
                end if
                last_node => last_node%next
            end do

            if (.not. is_associated) then
                cloned_route_head => clone_route_list(route_head)
                call find_routes(graph, neighbor, destination, cloned_route_head, all_routes,&
                    total_dist + edge%distance, total_print_mant + edge%print_mant, .false.)
            end if
            edge => edge%next_ari
        end do
    end if

        if (.not. is_route_added) then
            deallocate(new_node)
        end if
end subroutine find_routes
```

En esta subrutina en necesario la recursión para que busque todas las posibles rutas y es diferente al algoritmo de costo uniforme donde tenemos ya nodos visitados por eso es importante inicializar antes de hacer uso de esta subrutina. Posteriormente a eso la subrutina obtiene todas las rutas y las va aguardando en una lista la cual en la llamada add_Route es aquí donde se aguarda todas las posibles rutas.

Tabla hash

Los técnicos son vitales para el mantenimiento de las impresoras en cada sucursal. Por ello se utiliza una tabla hash donde se aguarda pero siempre y cuando esta table tiene su función de dispersión la cual es h(llv) = llv mod M , mediante esta fórmula la aplicación es capas de sabes donde irá en técnico aguardas, después se vio la necesidad de realizar una política de resolución de colisiones la cual será una doble dispersión por medio de la función descrita s(llv, i) = (mod (llv mod 7 + 1) * i, zise_table) mediante esta formula se da en el caso que tengamos problemas de insertar un técnico esta el la que realiza un colisión para posicionar en una nueva posición el técnico, se puede dar el caso que se llene la tabla para esto es importante realizar un rehashing donde se realizara cunado llegue al 70% de uso es decir cuando rebase este limite la tabla se duplicara para las nuevas inversiones de los técnicos.

La carga de técnicos en el sistema se realiza a través de un archivo en formato específico, que puede ser subido en la sección correspondiente de la interfaz. Este archivo debe seguir un formato predefinido que el sistema pueda leer e interpretar correctamente.

A continuación, se muestra la estructura como algunos subrutinas y funciones que fue necesario para elaborar la implementación de la tabla hash.

```
! Hash table structure
type techinical
    integer*8 :: dpi !key
    character(len=32) :: name
    character(len=32) :: last_name
    character(len=20) :: genre
    character(len=100) :: direccion
    integer :: phone
    integer :: jobs_done = 0 ! trabajos realizados
end type techinical


type hash
    integer :: elements
    integer :: size_table
    integer :: maxi ! porcent of table hash
    type(techinical), dimension(:), allocatable :: table ! table techinical
    contains
        procedure :: init !inicializar tabla
        procedure :: division ! encontrar indice
        procedure :: linear_probe ! Colicion (doble dispersion)
        procedure :: insert !insertar en la tabla
        procedure :: show !mostrar tabla hash
        procedure :: reashing
        procedure :: find_technician !Buscar tecnico si existe
        procedure :: graph_hash_table !Grafica de la tabla hash
        procedure :: find_technician_id !Buscar un tecnico es espesifico
        procedure :: list_all_technician !Listar tecnicos
end type hash
```

En la imagen posterior se muestra algunas de las políticas y funciones de dispersión la cuales son las importantes para el cálculo de la posición de un técnico a insertar

```
!Obtener la posicision
integer*8 function division(this, dpi)
    class(hash), intent(inout) :: this
    integer*8,  intent(in) :: dpi
    division = mod(dpi, this%size_table) ! dpi MOD size_table
end function division

!Obtener la posicion si hay una colision
integer*8 function linear_probe(this, dpi, i)
    class(hash), intent(inout) :: this
    integer*8, intent(in) :: dpi
    integer, intent(in) :: i ! contador de colisiones

    linear_probe = mod((mod(dpi, 7) + 1) * i, this%size_table)

end function linear_probe
```

Árbol de Merkle Y Block Chain

Para garantizar la seguridad e integridad de los datos de las rutas y mantenimientos realizados, el sistema implementa un Árbol de Merkle y una estructura de blockchain. Cada transacción o ruta optima será generada unas transacciones para esto es importante el árbol de Merkel para ello se utilizó una lista simple para la estructura del blockchain y para la estructura del árbol de Merkel se utilizó una lista simple pero con la peculiaridad que tendrá dos punteros como también es importante definir que este árbol es lleno por eso se estable que cuando no es binario el árbol se completa con -1 esto es para que el árbol sea lleno.
En la implementación del árbol de Merkle, se utilizan dos tipos de nodos: node_dataBloken y hash_node. Además, la estructura general del árbol se encapsula en el tipo tree_merkle es así cómo funciona la estructura

Nodos de Datos (node_dataBloken): Cada nodo de datos contiene un identificador único un valor de datos, y un puntero que enlaza con el siguiente nodo en la lista. Este tipo de nodo sirve para almacenar la información elemental que luego será referenciada en los nodos de hash del árbol.

Nodos Hash (hash_node): Cada nodo hash contiene un identificador único, una cadena de hash esto es principal explicar cuándo vamos a ser una transacción el árbol de Merkel juega una función importante la cual es realizar transacciones de manera que no haya un regreso por eso se codifica la información con ello será las transacción más seguras ya que hablamos del tema del blockchain esta parte del root Merkel es la que se pasa al block chain , y dos punteros (left y right) que apuntan a los nodos hijo izquierdo y derecho, respectivamente. También tiene un puntero (dataref) que referencia a un node_dataBloken para asociar los datos directamente con un nodo hash específico. Este diseño permite la

construcción del árbol de Merkle, donde cada nodo hash representa una combinación de sus nodos hijos o directamente los datos de las hojas.

Árbol de Merkle (tree_merkle): La estructura del árbol de Merkle se maneja con este tipo, que incluye punteros al nodo hash superior tophash y a los nodos de datos inicial y final data_head y data_coil. También mantiene un contador de posición pos, que es crucial durante la construcción del árbol. Contiene procedimientos para añadir datos, generar hashes, y operaciones de visualización y manipulación del árbol.

A continuación, se muestra imágenes donde se referencia la estructura del árbol de Merkel

```
!:::: Estructura Arbol Merkle
type node_dataBloken ! data node
    integer :: uid
    character(len = :), allocatable :: value
    type(node_dataBloken), pointer :: next => null()
end type node_dataBloken

type hash_node !nodo hash
    integer :: uid
    character(len = :), allocatable :: hash
    type(hash_node), pointer :: left => null()
    type(hash_node), pointer :: right => null()
    type(node_dataBloken), pointer :: dataref => null()
end type hash_node

type tree_merkle
    type(hash_node), pointer :: tophash => null()
    type(node_dataBloken), pointer :: data_head => null()
    type(node_dataBloken), pointer :: data_coil => null()
    integer :: pos = 0
    contains
        procedure :: size_list ! tamaño de mi lista nodos hojas
        procedure :: add ! insertar nodos hoja
        procedure :: create_tree_merkle ! creo arbol merkle
        procedure :: genhash !generar hash
        procedure :: datablok !obtengo los nodos hojas
        procedure :: showhash
        procedure :: show_dataBlock
        procedure :: generate
        procedure :: dot_tree_merkle
        procedure :: grap_Tree_merkle !graficar mi arbol merkle
        procedure :: get_root_hash
end type tree_merkle
!:::: end Estructura del arbol Merkle
```

A continuación, se muestra la subrutina que crea el árbol de Merkel con base al exponente esto lo hace descendentemente y para la creación del hash se utiliza la el 2 elevado al exponente y esto se crea de abajo para arriba.

```
!Se crea el arbol asignando espacio de acuerdo al expo
recursive subroutine create_tree_merkle(this, node, expo)
    class(tree_merkle), intent(inout) :: this
    type(hash_node), intent(inout), pointer :: node
    integer, intent(in) :: expo
    node%uid = uid
    uid = uid + 1
    if(expo > 0)then
        allocate(node%left)
        allocate(node%right)
        call this%create_tree_merkle(node%left, expo - 1)
        call this%create_tree_merkle(node%right, expo - 1)
    end if
end subroutine create_tree_merkle

!Se obtiene el genhash
recursive subroutine genhash(this, node, pow)
    class(tree_merkle), intent(inout) :: this
    type(hash_node), intent(inout) ,pointer :: node
    type(node_dataBloken), pointer :: leaf_node

    integer, intent(in) :: pow
    integer :: temp
    character(len=:), allocatable :: hash
        if(associated(node))then
            call this%genhash(node%left, pow) !postorden
            call this%genhash(node%right, pow)
            if(.not. associated(node%left) .and. .not. associated(node%right))then
                temp = pow - this%pos ! exp^3 => pow=8, this%pos = 8
                node%dataref => this%datablok(temp)
                this%pos = this%pos - 1
                hash = node%dataref%value
                node%hash = sha256(hash) !convertir a SHA256 la data
            else
                hash = adjustl(node%left%hash(1:len(node%left%hash)/2))// &
                adjustl(node%right%hash(1:len(node%right%hash)/2))
                node%hash = sha256(hash) !convertir a SHA256 la data
            end if
        end if
end subroutine genhash
```

Esta estructura integrada de cadena de bloques y árbol de Merkle proporciona un poderoso mecanismo de verificación y verificación de datos. Los árboles Merkle pueden verificar eficazmente los datos contenidos en cada bloque, mientras que el uso de bloques vinculados mediante hash garantiza que cualquier cambio en la información de un bloque sea rastreable, protegiendo así la cadena contra manipulaciones. La integración de las raíces de Merkle en cada bloque agrega una capa adicional de seguridad y eficiencia, lo que hace que la verificación y validación sean rápidas y confiables.

Para la implementación del blockchain se utilizo una lista simple enlazada la cual cuentas con:
Índice (index): Número secuencial del bloque dentro de la cadena.
Timestamp: Registro temporal del momento de creación del bloque.

Datablock: Contiene la carga útil del bloque, agrupada en una estructura all_data es decir que cuenta con la información de la ruta mas optimas
Nonce (once): Valor que se utiliza como valor de trabajo
Hash anterior (previus_hash): Referencia al hash del bloque anterior en la cadena, asegurando la integridad y la secuencia de la cadena.
Raíz de Merkle (root_merkle): Hash que representa la raíz del árbol de Merkle, consolidando todas las transacciones del bloque.
Hash del bloque (hash): Hash criptográfico del bloque que incluye el nonce, la raíz de Merkle, y elementos del bloque.

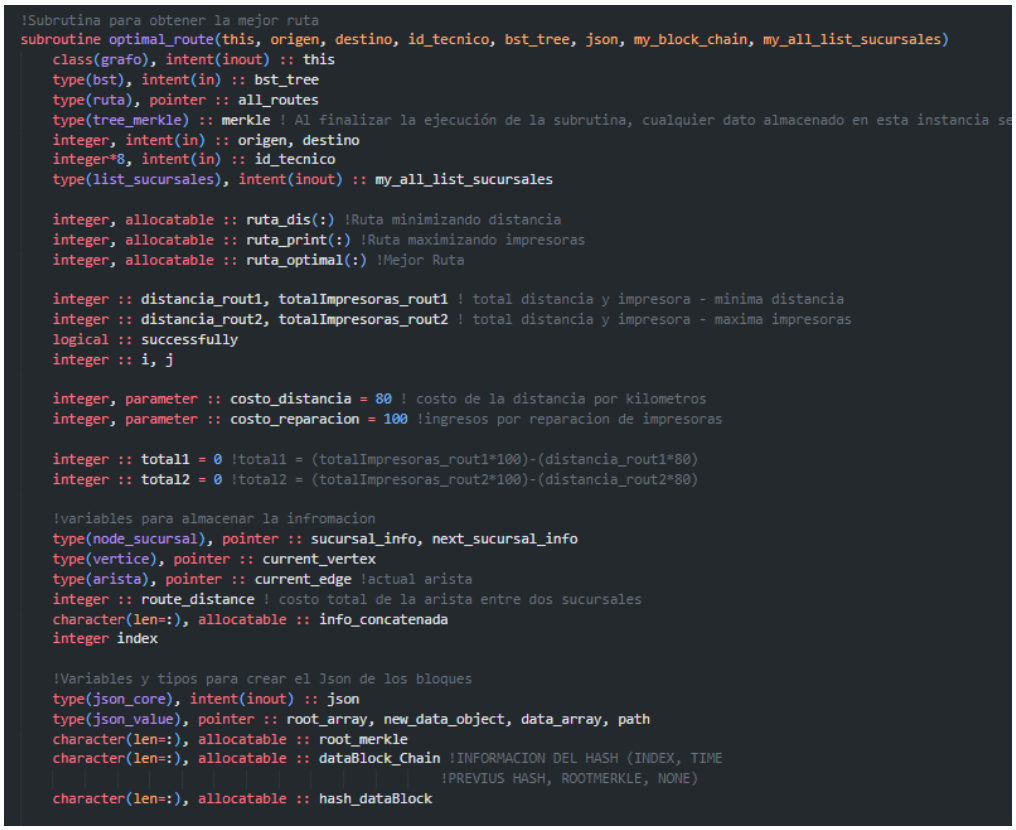
```
!:::: Estructuras BLOCK CHAIN
type data
    character(len=10) :: id_sucursal_o
    character(len=200) :: direccion_suc_o
    character(len=10) :: id_sucursal_d
    character(len=200) :: direccion_suc_d
    character(len=50) :: costo_suc
    type(data), pointer :: next_data => null()
end type data

type all_data
    type(data), pointer :: head_data => null()
    contains
        procedure :: add_data
end type

type node_DataBlock
    integer :: index
    character(len=30) :: timestamp
    type(all_data) :: datablock
    character(len=20) :: nonce
    character(len=:), allocatable :: previus_hash
    character(len=:), allocatable :: root_merkle
    character(len=:), allocatable :: hash
    type(node_DataBlock), pointer :: next_DataBlock => null()
end type node_DataBlock

type block_chain
    type(node_DataBlock), pointer :: head_dataBlock => null()
    contains
        procedure :: add_block !AÑADIR BLOCK CHAIN
        procedure :: show_block_chain
        procedure :: grap_block_chain !GRAFICAR BLOCK CHAIN
end type block_chain
```

Para acceder a la información del blockchain fue principal una función la cual es la misma que nos calcula cual es la ruta optima es aquí donde extraemos la información para después ingresarla a nuestro árbol merkle y después posterior a ello a nuestro block chain.



```
!Subrutina para obtener la mejor ruta
subroutine optimal_route(this, origen, destino, id_tecnico, bst_tree, json, my_block_chain, my_all_list_sucursales)
    class(grafo), intent(inout) :: this
    type(bst), intent(in) :: bst_tree
    type(ruta), pointer :: all_routes
    type(tree_merkle) :: merkle ! Al finalizar la ejecución de la subrutina, cualquier dato almacenado en esta instancia se
    integer, intent(in) :: origen, destino
    integer*8, intent(in) :: id_tecnico
    type(list_sucursales), intent(inout) :: my_all_list_sucursales

    integer, allocatable :: ruta_dis(:) !Ruta minimizando distancia
    integer, allocatable :: ruta_print(:) !Ruta maximizando impresoras
    integer, allocatable :: ruta_optimal(:) !Mejor Ruta

    integer :: distancia_rout1, totalImpresoras_rout1 ! total distancia y impresora - minima distancia
    integer :: distancia_rout2, totalImpresoras_rout2 ! total distancia y impresora - maxima impresoras
    logical :: successfully
    integer :: i, j

    integer, parameter :: costo_distancia = 80 ! costo de la distancia por kilometros
    integer, parameter :: costo_reparacion = 100 !ingresos por reparacion de impresoras

    integer :: total1 = 0 !total1 = (totalImpresoras_rout1*100)-(distancia_rout1*80)
    integer :: total2 = 0 !total2 = (totalImpresoras_rout2*100)-(distancia_rout2*80)

    !variables para almacenar la infromacion
    type(node_sucursal), pointer :: sucursal_info, next_sucursal_info
    type(vertice), pointer :: current_vertex
    type(arista), pointer :: current_edge !actual arista
    integer :: route_distance ! costo total de la arista entre dos sucursales
    character(len=:), allocatable :: info_concatenada
    integer index

    !Variables y tipos para crear el Json de los bloques
    type(json_core), intent(inout) :: json
    type(json_value), pointer :: root_array, new_data_object, data_array, path
    character(len=:), allocatable :: root_merkle
    character(len=:), allocatable :: dataBlock_Chain !INFORMACION DEL HASH (INDEX, TIME
                                                     !PREVIUS HASH, ROOTMERKLE, NONE)
    character(len=:), allocatable :: hash_dataBlock
```

Conclusiones

- Optimización de Rutas: Mediante el uso del algoritmo de costo uniforme y búsqueda por anchura, el sistema puede determinar no solo la ruta más corta entre sucursales basada en distancia, sino también aquellas que requieren un mayor número de interacciones, como el mantenimiento de impresoras. Esto subraya el compromiso del sistema con la eficiencia operativa y la respuesta rápida a las necesidades del negocio obteniendo la ruta con mayor ganancia.

- El sistema está diseñado para ser seguro y accesible solo para usuarios autorizados, como los administradores, quienes utilizan credenciales únicas para iniciar sesión. Esto asegura la integridad y la seguridad de la información manejada dentro de la aplicación.

- La aplicación utiliza estructuras de datos complejas, como árboles binarios de búsqueda (BST) para gestionar las sucursales y listas de adyacencia para mapear rutas entre estas. Esto facilita la implementación de algoritmos eficientes para la búsqueda y manejo de datos.

- Implementación de Blockchain y Árbol de Merkle: La integración de estas tecnologías no solo aumenta la seguridad de las transacciones dentro de la aplicación, sino que también asegura la integridad y la verificación de los datos de manera eficiente. Cada bloque en la cadena contiene información esencial codificada y verificada, lo que hace prácticamente imposible la manipulación de los datos sin que sea detectado.