Práctica 2

IBERO
Ciudad de México • Tijuana

Algoritmo de Dijkstra

Integrantes del equipo

César Mauricio Arellano Velásquez
Allan Jair Escamilla Hernández
Raúl González Portillo

Profesor

César Arturo Ángeles Ruiz

Materia

Taller de Desarrollo de Aplicaciones

Introducción:

El **algoritmo de Dijkstra**, también llamado **algoritmo de caminos mínimos**, es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista. Su nombre alude a Edsger Dijkstra, científico de la computación de los Países Bajos que lo describió por primera vez en 1959.

La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene. Se trata de una especialización de la búsqueda de costo uniforme y, como tal, no funciona en grafos con aristas de coste negativo (al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo).

Una de sus aplicaciones más importantes reside en el campo de la telemática. Gracias a él, es posible resolver grafos con muchos nodos, lo que sería muy complicado resolver sin dicho algoritmo, encontrando así las rutas más cortas entre un origen y todos los destinos en una red.

Algoritmo

Teniendo un grafo dirigido ponderado de N nodos no aislados, sea x el nodo inicial. Un vector D de tamaño N guardará al final del algoritmo las distancias desde x hasta el resto de los nodos.

1. Inicializar todas las distancias en D con un valor infinito relativo, ya que son desconocidas al principio, exceptuando la de x, que se debe colocar en 0, debido a que la distancia de x a x sería 0.
2. Sea a = x (Se toma *a* como nodo actual.)
3. Se recorren todos los nodos adyacentes de *a*, excepto los nodos marcados. Se les llamará *nodos no marcados* v_i.
4. Para el nodo actual, se calcula la distancia tentativa desde dicho nodo hasta sus vecinos con la siguiente fórmula: $dt(v_i) = D_a + d(a,v_i)$. Es decir, la distancia tentativa del nodo 'v_i' es la distancia que actualmente tiene el nodo en el vector D más la distancia desde dicho nodo 'a' (el actual) hasta el nodo v_i. Si la distancia tentativa es menor que la distancia almacenada en el vector, entonces se actualiza el vector con esta distancia tentativa. Es decir, si $dt(v_i) < D_{vi} \rightarrow D_{vi} = dt(v_i)$
5. Se marca como completo el nodo a.
6. Se toma como próximo nodo actual el de menor valor en D (puede hacerse almacenando los valores en una cola de prioridad) y se regresa al paso 3, mientras existan nodos no marcados.

Una vez terminado al algoritmo, D estará completamente lleno.

Objetivos:

- Comprender el funcionamiento del algoritmo Dijkstra con el fin de encontrar la ruta más corta de un nodo origen a un nodo destino.
- Entender la lógica con la cual plataformas como Google Maps / Waze calculan las rutas más eficientes.

Análisis

Pseudocódigo:

```
#define return -> retorna el valor de alguna operación o variable.
#define free -> Liberar Memoria.
#define exit(0) -> Finaliza la ejecución del programa.
#define sizeof() -> Retorna el valor de un tipo de dato en bytes.

Principal()
{
    Raiz = NULL;
    opcion = 2, i = 0, cuentaNodos = 0;
    etiqueta = 'a';
    visitados = 0;
    cuentaPasos = 0;
    bandera = 0;
    Hacer{
        Imprimir("Ingresar la cantidad de conexiones por nodo->");
        Leer(conexiones);
        Si (conexiones < 1)
            Imprimir("Los nodos deben tener al menos una conexión");
    } mientras (conexiones < 1);
    hacer{
        menu();
        Leer(opcion);
        Selección (opcion){
        caso 1:
            insertarNodo(etiqueta, conexiones, i | Raiz);
            etiqueta++;
            i++;
            cuentaNodos++;
            Imprimir("Se ha insertado correctamente el nodo al grafo!");
            romper;
        caso 2:
            Si(Raiz <> NULL)
                agregarArista(conexiones | Raiz);
            Si no
                Imprimir("El grafo esta vacio\n");
            romper;
        case 3:
            Si(Raiz <> NULL)
```

```
                buscarDato(Raiz, conexiones | );
            Si no
                Imprimir("El grafo esta vacio");
            romper;
        caso 4:

            Si(Raiz <> NULL)
                desplegarGrafo(cuentaNodos | );
            Si no
                Imprimir("El grafo esta vacio");
            romper;
        caso 5:
            Si(Raiz <> NULL){
                inicializarTabla(cuentaNodos | t1);
                Dijkstra(cuentaPasos,  Raiz->etiqueta,  cuentaNodos,  conexiones,
visitados | t1, Raiz);
                Imprimir("Tabla calculada");
            }
            Si no
                Imprimir("El grafo esta vacío");
            bandera = 1;
            romper;
        caso 6:
            Si(bandera = 0)
                Imprimir("La tabla aun no ha sido calculada");
            Si no
                deplegarTabla(t1, cuentaNodos | );
            romper;
        caso 7:
            Si(bandera = 1){
                calcular_ruta(t1, cuentaNodos | );
            }
            Si no
                Imprimir("Aun no se ha calculado la tabla");
            romper;
        caso 8:
            Si(Raiz <> NULL){
                liberar_memoria(0, conexiones, NULL | Raiz);
            }
            Imprimir("Saliendo del programa...");
            exit(0);
            romper;
        Por defecto:
            Imprimir("Ingresa una opción valida!");
            romper;
        }
        Imprimir("Presione enter para continuar...");
    } mientras (opcion <> 8);
}
insertarNodo(etiqueta,conexiones, i | Raiz){
    aux = NULL;
    new(temp);
    ↑temp.etiqueta = etiqueta;
```

```
    temp->cantidadConexiones = 0;
    ↑temp.coincidencias = 0;
    new(↑temp.conexiones);
    new(↑temp->costo);
    ↑temp.next = NULL;
    Desde i = 0; Hasta i < conexiones; i++
        ↑temp.conexiones[i] = NULL;
    Si(Raiz = NULL)
        Raiz = temp;
    Si no{
        Hacer{
            Imprimir(“Ingresar el nodo al que va conectado: ");
            Leer(NodoConnect);
            buscarNodo(buscado | Raiz, aux)
            Si (aux = NULL)
                Imprimir("El nodo ingresado no existe en el grafo, asegúrese de
ingresar uno que realmente exista!\n\n");
        } mientras (aux = NULL);
        Si(↑aux.cantidadConexiones < conexiones){
            Imprimir("Ingresar el costo: ");
            Leer(↑aux.costo[↑aux.cantidadConexiones]));
            ↑temp.costo[↑temp.cantidadConexiones] =
↑aux.costo[↑aux.cantidadConexiones];
            ↑aux.conexiones[↑aux.cantidadConexiones] = temp;
            ↑temp.conexiones[↑temp.cantidadConexiones] = aux;
            (↑aux.cantidadConexiones)++;
            (↑temp.cantidadConexiones)++;
            Si(i <> 0){
                temp2 = Raiz;
                mientras(↑temp2.next <> NULL)
                    temp2 = ↑temp2.next;
                ↑temp2.next = temp;
            }
        }Si no{
            Imprimir("Ya no hay espacio para más conexiones");
        }
    }
}

buscarNodo(Raiz, buscado | aux){
    temp = Raiz;
    mientras (temp <> NULL){
        Si(temp <> NULL AND ↑temp.etiqueta = buscado)
            aux = temp;
        temp = ↑temp.next;
    }
}
```

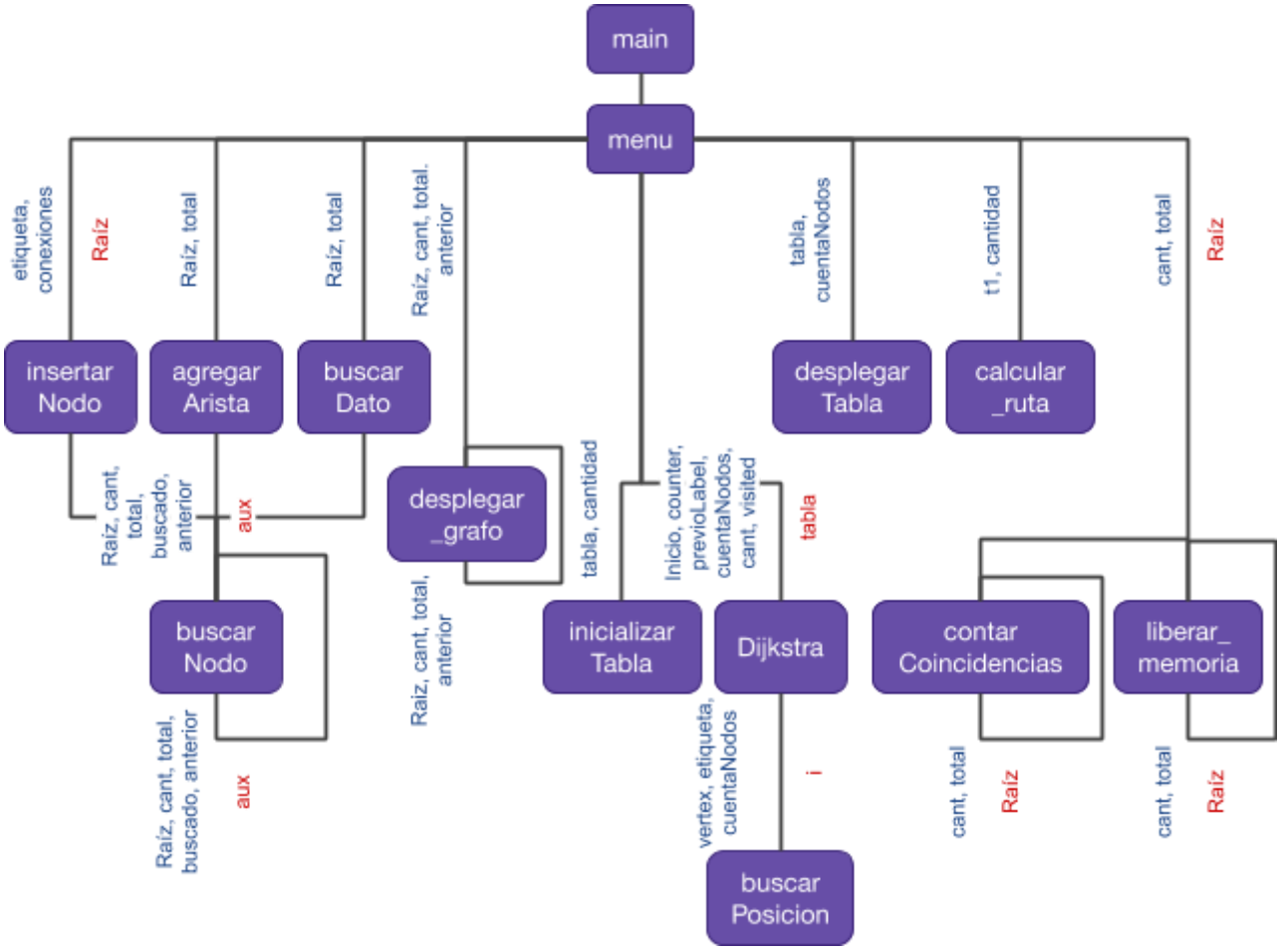
```
buscarDato(Raiz, total | ){
    aux = NULL;
    Imprimir("Ingresar nodo a buscar-> ");
    Leer(etiqueta);
    buscarNodo(Raiz, buscado | aux)
    Si(aux = NULL){
        Imprimir("No se ha encontrado el dato");
        return;
    }
    Imprimir("El dato”+aux->etiqueta+“ha sido encontrado en el grafo,Sus
conexiones son: ");
    Desde i = 0; Hasta i < total; i++
        Si( ↑aux.conexiones[i] <> NULL)

Imprimir(“Nodo:”+ ↑aux ↑conexiones[i].etiqueta+“Costo:"+ ↑aux.costo[i]);
        Si no
            Imprimir("Nodo: NULL, Costo: NULL");
}

agregarArista(total | Raiz){
    agregarArista = NULL;
    recibirConexion = NULL;
    Imprimir("Ingresar el nodo al que desea agregar la arista-> ");
    Leer(etiqueta1);
    buscarNodo(Raiz, etiqueta1, | agregarArista);
    Imprimir("Ingresar el nodo al que se va a conectar el nodo anterior-> ");
    Leer(etiqueta2);
    buscarNodo(Raiz, etiqueta2, | recibirConexion);
    Si(agregarArista = NULL OR recibirConexion = NULL){
        Imprimir("Alguno de los nodos ingresados no existe\n");
        return;
    }
    Si( ↑agregarArista.cantidadConexiones >= total OR
↑recibirConexion.cantidadConexiones >= total){
        Imprimir("Alguno de los nodos ya no tiene espacio para más conexiones");
        return;
    }
    Imprimir("Ingresar costo del salto: ");
    Leer( ↑agregarArista.costo[ ↑agregarArista.cantidadConexiones]));
     ↑recibirConexion.costo[ ↑recibirConexion.cantidadConexiones] =
↑agregarArista.costo[ ↑agregarArista.cantidadConexiones];
     ↑agregarArista.conexiones[ ↑agregarArista.cantidadConexiones] =
recibirConexion;
     ↑recibirConexion.conexiones[ ↑recibirConexion.cantidadConexiones] =
agregarArista;
     ↑agregarArista.cantidadConexiones = ↑agregarArista.cantidadConexiones + 1;
     ↑recibirConexion.cantidadConexiones = ↑recibirConexion.cantidadConexiones
+ 1;
```

```
    Imprimir("Se ha agregado la arista con éxito");
}


liberar_memoria(cant,total,anterior | Raiz){
    aux = Raiz;
    mientras (Raiz <> NULL){
        aux = Raiz;
        Raiz = ↑Raiz.next;
        free(aux);
    }
}

deplegarTabla(tabla, cuentaNodos | ){
    Imprimir("VERTEX: ");
    Desde i = 0; Hasta i < cuentaNodos; i++
        Imprimir(tabla.vertex[i]);

    Imprimir("SDF: ");
    Desde i = 0; Hasta i < cuentaNodos; i++
        Imprimir(tabla.sdf[i]);

    Imprimir("PREV VERTEX: ");
    Desde i = 0; Hasta i < cuentaNodos; i++
        Imprimir(tabla.prevVertex[i]);
}

void desplegarGrafo(cuentaNodos | ){
    lbl = 'a';
    Desde i = 0; Hasta i < cuentaNodos; i++{
        Imprimir(lbl);
        lbl++;
    }
}
void inicializarTabla(cantidad | tabla){
    init = 'a';
    new(↑tabla.sdf);
    new(↑tabla.vertex);
    new(↑tabla.prevVertex);
    Desde i = 0;  Hasta i < cantidad; i++{
         ↑tabla.vertex[i] = init;
         ↑tabla.sdf[i] = 100000;
        init++;
    }
}

Dijkstra(counter, previoLabel,cuentaNodos, cant, visited | Inicio,tabla){
    contador = 0;
    temp = 0;
    bits = sizeof(unsigned int) * 8 - cuentaNodos;
    mientras (contador < cant){
```

```
            posicion = buscarPosicion( ↑tabla.vertex,  ↑Inicio.etiqueta,
cuentaNodos | );
            temp2 = visited;
            temp = 1;
            temp <<= bits + posicion;
            visited = visited OR temp;
            Si(visited <> temp2){
                Si (counter <  ↑tabla.sdf[posicion]){
                     ↑tabla.sdf[posicion] = counter;
                     ↑tabla.prevVertex[posicion] = previoLabel;
                }
                Si ( ↑Inicio.conexiones[contador] <> NULL){
                    Dijkstra(counter + Inicio->costo[contador],
Inicio->etiqueta, cuentaNodos, cant, visited |  ↑Inicio.conexiones[contador],
tabla);
                    visited = temp2;
                }
            }
        contador++;
    }
}

buscarPosicion(vertex etiqueta, cuentaNodos | ){
    desde i = 0; Hasta i < cuentaNodos; i++
        Si(vertex[i] = etiqueta)
            return i;
}

calcular_ruta(t1,cantidad | ){
    p = 0, min = 0;
    Imprimir("Ingresar el nodo de la red al que desea llegar-> ");
    Leer(nodo);
    Desde i = 0; Hasta i < cantidad; i++
        Si(t1.vertex[i] = nodo){
            p = 1;
            min = t1.sdf[i];
        }
    Si(p = 0){
        Imprimir("El nodo no se encuentra en la red):");
        return;
    }
    Imprimir("La ruta mas corta es: ");
    etiqueta = nodo;
    Imprimir(etiqueta);
    mientras (etiqueta <> 'a'){
        desde i = 0; Hasta i < cantidad; i++
            Si (t1.vertex[i] = etiqueta){
                etiqueta = t1.prevVertex[i];
                romper;
            }
        Imprimir(etiqueta);
```

```
    }
    Imprimir("El costo total de este camino es de:"+min);
}
```

Diagrama IPO



Entradas Procesos y Salidas

Entradas

Nombre	Descripción
Raíz	Almacena la dirección del nodo Raíz del grafo.
etiqueta	Guarda la letra de la etiqueta del nodo a operar, es utilizada en varias funciones, véase **insertarNodo**, **agregarArista**, etc.
conexiones	Guarda la cantidad máxima de conexiones por nodo, es utilizada en casi todas las funciones.

cant	Contador que almacena las veces que se accede a cada conexión de un nodo.
buscado	Utilizada en **buscarNodo** y con una función similar a etiqueta, esta variable guarda la etiqueta del nodo a buscar.
counter	Guarda los costos de llegar de un nodo a otro.
previoLabel	Guarda el label anterior al que se esté operando en determinado momento.
cuentaNodos	Cuenta y almacena la cantidad de nodos que hay en el grafo.

Procesos

Nombre	**Descripción**
insertarNodo	Inserta un nodo al grafo, si este es el primero su dirección se guardará en Raíz, de lo contrario el usuario determinará a que Nodo se conectará.
agregarArista	Agrega una arista a un nodo del grafo determinado por el usuario.
buscarDato	Busca un dato en el grafo determinado por el usuario.
buscarNodo	Parte de **buscarDato**, esta función busca la dirección de memoria del Nodo especificado por el usuario.
inicializarTabla	Inicializa una nueva tabla SDF vacía.
Dijkstra	Implementa el algoritmo de Dijkstra.
buscarPosicion	Busca la posición de un nodo dentro de la tabla SDF.
desplegarTabla	Imprime la tabla SDF calculada por Dijkstra.
calcular_ruta	Calcula la ruta más corta desde un nodo especificado por el usuario basándose en la tabla SDF
contarCoincidencias	Cuenta las veces en las que se puede

	llegar a un mismo nodo.
liberar_memoria	Libera la memoria que se utilizó para generar el grafo.

Salidas

Nombre	**Descripción**
Raiz	Actualiza la dirección de memoria del nodo raíz del grafo en determinadas funciones como **liberar_memoria.**
aux	Devuelve la dirección de memoria del nodo buscado en la función **buscarNodo.**
Tabla	Almacena y devuelve la tabla de SDF para determinado nodo en forma de estructura tipo Tabla.
Tabla.sdf[]	Parte de la estructura Tabla, es un arreglo que guarda la columna “SDF” de la tabla SDF.
Tabla.vertex[]	Parte de la estructura Tabla, es un arreglo que guarda la columna de los Vértices de la tabla SDF.
Tabla.prevVertex[]	Parte de la estructura Tabla, es un arreglo que guarda la columna “Vértice anterior” de la tabla SDF.
Tabla.visited	Parte de la estructura Tabla, es un entero sin símbolo que guarda qué nodos se han visitado por medio de operaciones bitwise.

Código

Archivo dijkstra.h

```
/*
 * @author:   Allan Jair Escamilla Hernández, Raúl González Portillo y César
Mauricio Arellano Velásquez
 * @date:    12/octubre/2019
 * @file:    dijkstra.c
 * @brief:  Implementación del algoritmo Dijkstra a un grafo.
 */

// INCLUYENDO LAS BIBLIOTECAS A UTILIZAR
#include<stdio.h>
#include<stdlib.h>
```

```
// DEFINIENDO LA ESTRUCTURA DE CADA NODO DEL GRAFO
typedef struct defNodo{
    char etiqueta;
    int cantidadConexiones, coincidencias;
    int *costo;
    struct defNodo **conexiones;
    struct defNodo* next;
} Nodo;

// DEFINIENDO ESTRUCTURA DE LA TABLA DONDE SE GUARDARAN LOS CAMINOS MAS CORTOS
typedef struct defTabla{
    int *sdf;
    char *vertex;
    char *prevVertex;
    int counter;
    unsigned int visited;
}Tabla;

// Estructuras de las funciones
void Dijkstra(Nodo *Inicio, Tabla *tabla, int counter, char previoLabel, int
cuentaNodos, int cant, unsigned int visited);
void inicializarTabla(Tabla *tabla, int cantidad);
int buscarPosicion(char *vertex, char etiqueta, int cuentaNodos);
void calcular_ruta(Tabla t1, int cantidad);



/* * Funcion que inicializa la tabla que contendrá los caminos.
    * @param Tabla* tabla recibe la tabla donde se calcularán los caminos más
cortos.
   * @param int cantidad recibe la cantidad de nodos en el grafo.
*/
void inicializarTabla(Tabla* tabla, int cantidad){
    char init = 'a';
    tabla->sdf = (int*)malloc(sizeof(int) * cantidad);
    tabla->vertex = (char*)malloc(sizeof(char) * cantidad);
    tabla->prevVertex = (char*)malloc(sizeof(char) * cantidad);
    for(int i = 0; i < cantidad; i++){
        tabla->vertex[i] = init;
        tabla->sdf[i] = 100000;
        init++;
    }
}

/* * Función que implementa el algoritmo de Dijkstra.
   * @param Nodo* Inicio recibe el nodo inicial del grafo.
    * @param Tabla tabla recibe la tabla donde se calcularon los caminos más
cortos.
   * @param int counter recibe una variable que cuenta los costos de llegar a un
nodo u otro.
    * @param char previoLabel recibe el label anterior al nodo actual en el
grafo.
```

```
   * @param int cuentaNodos recibe la cantidad de nodos en el grafo.
   * @param int cant es la cantidad de conexiones que tiene cada nodo.
    * @param unsigned int visited recibe los nodos que ya han sido visitados en el grafo.
*/
void Dijkstra(Nodo *Inicio, Tabla *tabla, int counter, char previoLabel, int cuentaNodos, int cant, unsigned int visited){
    int contador = 0;
    unsigned int temp = 0, temp2;
    int bits = sizeof(unsigned int) * 8 - cuentaNodos;
    while (contador < cant){
                int posicion = buscarPosicion(tabla->vertex, Inicio->etiqueta, cuentaNodos);
            temp2 = visited;
            temp = 1;
            temp <<= bits + posicion;
            visited |= temp;
            if(visited != temp2){
                if (counter < tabla->sdf[posicion]){
                    tabla->sdf[posicion] = counter;
                    tabla->prevVertex[posicion] = previoLabel;
                }
                if (Inicio->conexiones[contador] != NULL){
                        Dijkstra(Inicio->conexiones[contador], tabla, counter + Inicio->costo[contador], Inicio->etiqueta, cuentaNodos, cant, visited);
                    visited = temp2;
                }
            }
        contador++;
    }
}

/* * Función que busca la posición de un nodo dentro de la tabla.
    * @param char vertex recibe un arreglo de vértices donde se va a buscar un nodo.
   * @param char etiqueta recibe el valor que se va a buscar.
   * @param int cuentaNodos recibe la cantidad de nodos que hay en el grafo.
*/
int buscarPosicion(char *vertex, char etiqueta, int cuentaNodos){
    for(int i = 0; i < cuentaNodos; i++)
        if(vertex[i] == etiqueta)
            return i;
}

/* * Función que calcula la ruta mas corta basándose en la tabla que se ha calculado anteriormente.
   * @param Tabla t1 recibe la tabla con los caminos más cortos.
    * @param int cantidad recibe la cantidad de conexiones que tiene cada nodo dentro del grafo.
*/
void calcular_ruta(Tabla t1, int cantidad){
    char nodo, etiqueta;
    int p = 0, min = 0;
```

```
    printf("Ingresar el nodo de la red al que desea llegar-> ");
    __fpurge(stdin);
    scanf("%c", &nodo);
    for(int i = 0; i < cantidad; i++)
        if(t1.vertex[i] == nodo){
            p = 1;
            min = t1.sdf[i];
        }
    if(p == 0){
        printf("El nodo no se encuentra en la red):\n");
        return;
    }
    printf("La ruta más corta es: ");
    etiqueta = nodo;
    printf("%c <- ", etiqueta);
    while (etiqueta != 'a'){
        for(int i = 0; i < cantidad; i++)
            if(t1.vertex[i] == etiqueta){
                etiqueta = t1.prevVertex[i];
                break;
            }
        printf("%c <- ", etiqueta);
    }
    printf("\nEl costo total de este camino es de %d\n", min);
}
```

Archivo main.c

```
/*
 * @author:  Allan Jair Escamilla Hernández, Raúl González Portillo y César
Mauricio Arellano Velásquez
 * @date:    17/septiembre/2019
 * @file:    grafos.c
 * @brief:  Implementación de un generador de grafos y el algoritmo de Dijkstra.
 * -> Con una estructura de datos en C generar un tipo de datos "nodo" que
contenga n conexiones definidas por el usuario.
 * -> Implementar las funciones de : anadir_nodo, anadir_vertice,
desplegar_grafo, buscar_dato.
 * -> Cada nodo será etiquetado por su orden de inserción.
 */
#include"dijkstra.h" // Incluimos el archivo que contiene las estructuras y la
implementación del algoritmo de Dijkstra



// Prototipos de las funciones
void insertarNodo(Nodo** Raiz, char etiqueta, int conexiones, int i);
void buscarNodo(Nodo *Raiz, char buscado, Nodo **aux);
void agregarArista(Nodo *Raiz, int total);
void buscarDato(Nodo *Raiz, int total);
void liberar_memoria(Nodo** Raiz, int cant, int total, Nodo* anterior);
void deplegarTabla(Tabla tabla, int cuentaNodos);
void desplegarGrafo(int cuentaNodos);
void menu();
```

```
// Función principal
int main(){
    // DEFINIENDO VARIABLES
    Nodo* Raiz = NULL, *Inicial;
    int opcion = 2, conexiones, i = 0, cuentaNodos = 0;
    char etiqueta = 'a';
    unsigned int visitados = 0;
    int cuentaPasos = 0;
    int bandera = 0;
    Tabla t1;
    // CONTINUACIÓN DE LA EJECUCIÓN DEL PROGRAMA
    do{
        printf("Ingresar la cantidad de conexiones por nodo-> ");
        scanf("%d", &conexiones);
        if (conexiones < 1)
            printf("Los nodos deben tener al menos una conexión\n");
    } while (conexiones < 1);
    do{
        menu(); // DESPLEGANDO MENU
        scanf("%d", &opcion);
        switch (opcion){
        case 1:
            insertarNodo(&Raiz, etiqueta, conexiones, i);
            etiqueta++;
            i++;
            cuentaNodos++;
            printf("Se ha insertado correctamente el nodo al grafo!\n\n");
            break;
        case 2:
            if(Raiz != NULL)
                agregarArista(Raiz, conexiones);
            else
                printf("El grafo está vacío\n");
            break;
        case 3:
            if(Raiz != NULL)
                buscarDato(Raíz, conexiones);
            else
                printf("El grafo está vacío\n");
            break;
        case 4:

            if(Raiz != NULL)
                desplegarGrafo(cuentaNodos);
            else
                printf("El grafo esta vacio\n");
            break;
        case 5:
            if(Raiz != NULL){
                inicializarTabla(&t1, cuentaNodos);
                Dijkstra(Raiz, &t1, cuentaPasos, Raiz->etiqueta, cuentaNodos,
conexiones, visitados);
```

```
                printf("Tabla calculada\n");
            }else
                printf("El grafo está vacío\n");
            bandera = 1;
            break;
        case 6:
            if(bandera == 0)
                printf("La tabla aún no ha sido calculada\n");
            else
                deplegarTabla(t1, cuentaNodos);
            break;
        case 7:
            if(bandera == 1){
                calcular_ruta(t1, cuentaNodos);
            }else
                printf("Aun no se ha calculado la tabla\n");
            break;
        case 8:
            if(Raiz != NULL){
                liberar_memoria(&Raiz, 0, conexiones, NULL);
            }
            printf("Saliendo del programa... \n");
            exit(0);
            break;
        default:
            printf("Ingresa una opcion valida!\n");
            break;
        }
        printf("\nPresione enter para continuar... ");
        __fpurge(stdin);
        getchar();
    } while (opcion != 8);

    return 0;
}

// DESARROLLANDO LAS FUNCIONES

/* * Función que imprime el menú
*/
void menu(){
    system("clear");
    printf("1.- Insertar nodo.\n");
    printf("2.- Agregar arista.\n");
    printf("3.- Buscar dato.\n");
    printf("4.- Desplegar grafo.\n");
    printf("5.- Calcular tabla\n");
    printf("6.- Desplegar tabla.\n");
    printf("7.- Calcular ruta más corta.\n");
    printf("8.- Salir.\n");
    printf("\n\nSeleccione una opción-> ");
}
```

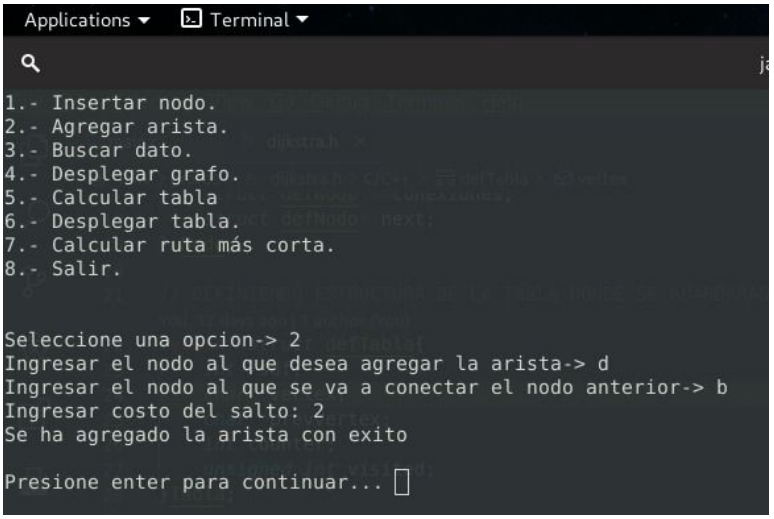
```
/* * Función que inserta un nodo al grafo.
   * @param Nodo** Raíz recibe la dirección de memoria del nodo raíz del grafo.
   * @param char etiqueta recibe un valor que contendrá la etiqueta del nodo a
insertar.
   * @param int conexiones recibe la cantidad de conexiones por nodo.
*/
void insertarNodo(Nodo **Raiz, char etiqueta, int conexiones, int i){
    char NodoConnect;
    Nodo* aux = NULL,  *temp2;
    Nodo* temp = (Nodo*)malloc(sizeof(Nodo));
    temp->etiqueta = etiqueta;
    temp->cantidadConexiones = 0;
    temp->coincidencias = 0;
    temp->conexiones = (Nodo**)malloc(sizeof(Nodo*) * conexiones);
    temp->costo = (int*)malloc(sizeof(int)*conexiones);
    temp->next = NULL;
    for(int i = 0; i < conexiones; i++)
        temp->conexiones[i] = NULL;
    if(*Raiz == NULL)
        *Raiz = temp;
    else{
        do{
            printf("Ingresar el nodo al que va conectado: ");
            scanf(" %c", &NodoConnect);
            buscarNodo(*Raiz, NodoConnect, &aux);
            if (aux == NULL)
                printf("El nodo ingresado no existe en el grafo, asegúrese de
ingresar uno que realmente exista!\n\n");
        } while (aux == NULL);
        if(aux->cantidadConexiones < conexiones){
            printf("Ingresar el costo: ");
            scanf("%d", &(aux->costo[aux->cantidadConexiones]));
            temp->costo[temp->cantidadConexiones] =
aux->costo[aux->cantidadConexiones];
            aux->conexiones[aux->cantidadConexiones] = temp;
            temp->conexiones[temp->cantidadConexiones] = aux;
            (aux->cantidadConexiones)++;
            (temp->cantidadConexiones)++;
            if(i != 0){
                temp2 = *Raiz;
                while(temp2->next != NULL)
                    temp2 = temp2->next;
                temp2->next = temp;
            }
        }else{
            printf("Ya no hay espacio para más conexiones\n");
        }
    }
}

/* * Función que busca un nodo dado por el usuario
   * @param Nodo* Raiz recibe la dirección de memoria del nodo raíz del grafo.
```

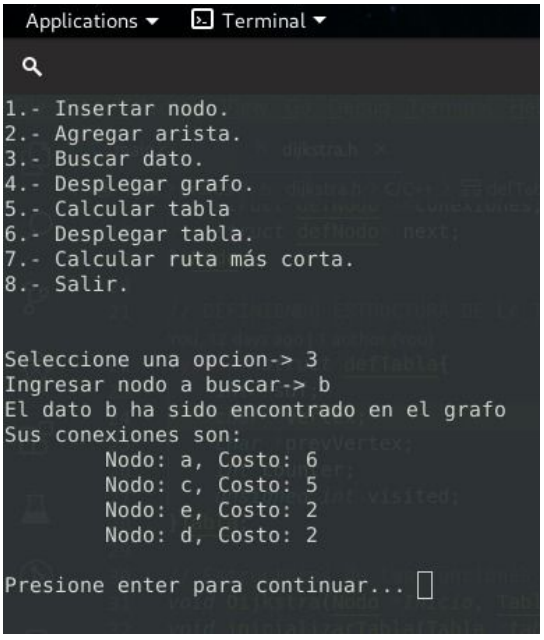
```
   * @param int cant recibe una variable que contara las veces que se acceda a
cada conexión de un nodo.
   * @param int total recibe el total de conexiones por nodo.
   * @param char buscado recibe la etiqueta del nodo a buscar.
   * @param Nodo** aux recibe la dirección de memoria de un apuntador que
almacenará el nodo buscado.
*/
void buscarNodo(Nodo* Raiz, char buscado, Nodo** aux){
    Nodo* temp = Raiz;
    while (temp != NULL){
        if(temp != NULL && temp->etiqueta == buscado)
            *aux = temp;
        temp = temp->next;
    }
}

/* * Función que busca un dato en el grafo.
   * @param Nodo* Raiz recibe la dirección de memoria del nodo raíz del grafo.
   * @param int total recibe el total de conexiones por nodo.
*/
void buscarDato(Nodo* Raiz, int total){
    char etiqueta;
    Nodo* aux = NULL;
    printf("Ingresar nodo a buscar-> ");
    __fpurge(stdin);
    scanf("%c", &etiqueta);
    buscarNodo(Raiz, etiqueta, &aux);
    if(aux == NULL){
        printf("No se ha encontrado el dato\n");
        return;
    }
    printf("El dato %c ha sido encontrado en el grafo\nSus conexiones son: \n",
aux->etiqueta);
    for(int i = 0; i < total; i++)
        if(aux->conexiones[i] != NULL)
            printf("\tNodo: %c, Costo: %d\n", aux->conexiones[i]->etiqueta,
aux->costo[i]);
        else
            printf("\tNodo: NULL, Costo: NULL\n");
}

/* * Función que agrega una arista al grafo.
   * @param Nodo* Raiz recibe la dirección de memoria del nodo raíz del grafo.
   * @param int total recibe el total de conexiones por nodo.
*/
void agregarArista(Nodo* Raiz, int total){
    char etiqueta1, etiqueta2;
    Nodo* agregarArista = NULL, *recibirConexion = NULL;
    printf("Ingresar el nodo al que desea agregar la arista-> ");
    __fpurge(stdin);
    scanf("%c", &etiqueta1);
    buscarNodo(Raiz, etiqueta1, &agregarArista);
    printf("Ingresar el nodo al que se va a conectar el nodo anterior-> ");
```

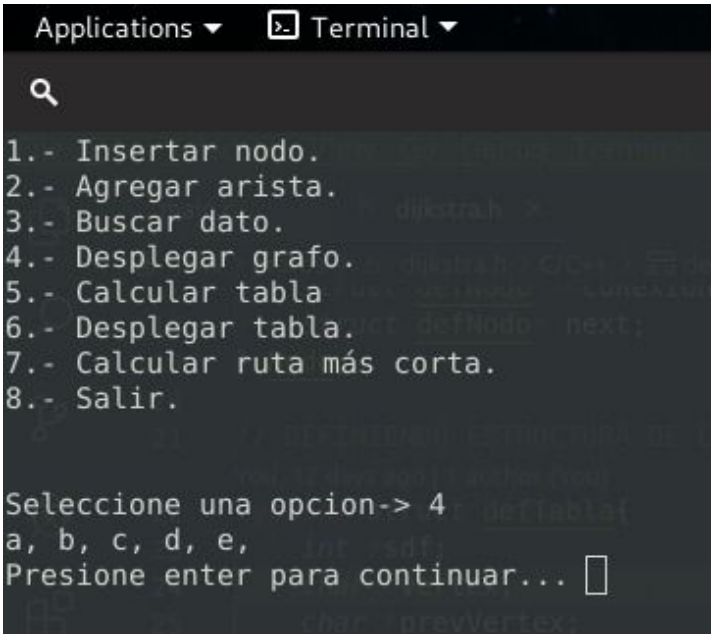
```
    __fpurge(stdin);
    scanf("%c", &etiqueta2);
    buscarNodo(Raiz, etiqueta2, &recibirConexion);
    if(agregarArista == NULL || recibirConexion == NULL){
        printf("Alguno de los nodos ingresados no existe\n");
        return;
    }
    if(agregarArista->cantidadConexiones >= total ||
recibirConexion->cantidadConexiones >= total){
        printf("Alguno de los nodos ya no tiene espacio para más conexiones\n");
        return;
    }
    printf("Ingresar costo del salto: ");
    scanf("%d", &(agregarArista->costo[agregarArista->cantidadConexiones]));
    recibirConexion->costo[recibirConexion->cantidadConexiones] =
agregarArista->costo[agregarArista->cantidadConexiones];
    agregarArista->conexiones[agregarArista->cantidadConexiones] =
recibirConexion;
    recibirConexion->conexiones[recibirConexion->cantidadConexiones] =
agregarArista;
    agregarArista->cantidadConexiones = agregarArista->cantidadConexiones + 1;
    recibirConexion->cantidadConexiones = recibirConexion->cantidadConexiones +
1;
    printf("Se ha agregado la arista con éxito\n");
}


/* * Función que libera la memoria del grafo.
   * @param Nodo** Raiz recibe la direccion de memoria del nodo raíz del grafo.
   * @param int cant recibe una variable que contara las veces que se acceda a
cada conexión de un nodo.
   * @param int total recibe el total de conexiones por nodo.
*/
void liberar_memoria(Nodo** Raiz, int cant, int total, Nodo* anterior){
    Nodo* aux = *Raiz;
    while (*Raiz != NULL){
        aux = *Raiz;
        *Raiz = (*Raiz)->next;
        free(aux);
    }
}

/* * Función que despliega la tabla de calculada por el algoritmo de Dijkstra.
   * @param Tabla tabla recibe la tabla donde se calcularon los caminos más
cortos.
   * @param int cuentaNodos recibe la cantidad de nodos en el grafo.
*/
void deplegarTabla(Tabla tabla, int cuentaNodos){
    printf("VERTEX: ");
    for(int i = 0; i < cuentaNodos; i++)
        printf("%c, ", tabla.vertex[i]);

    printf("\nSDF: ");
```

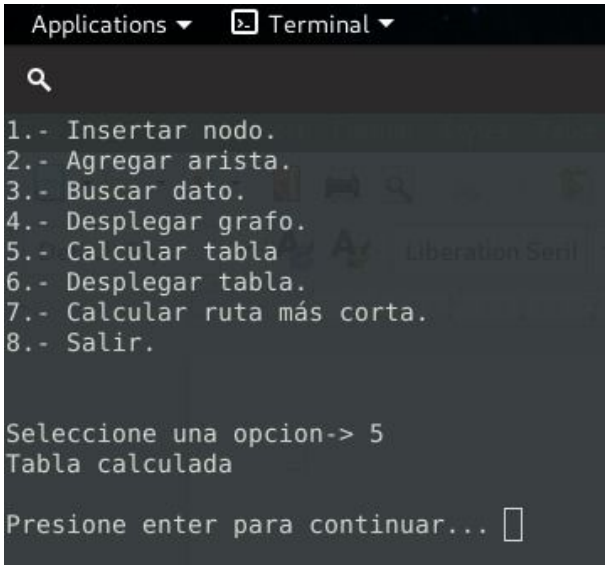
```
    for (int i = 0; i < cuentaNodos; i++)
        printf("%d, ", tabla.sdf[i]);

    printf("\nPREV VERTEX: ");
    for (int i = 0; i < cuentaNodos; i++)
        printf("%c, ", tabla.prevVertex[i]);
}

/* * Función que despliega las etiquetas de los nodos pertenecientes al grafo.
   * @param int cuentaNodos recibe la cantidad de nodos en el grafo.
*/
void desplegarGrafo(int cuentaNodos){
    char lbl = 'a';
    for(int i = 0; i < cuentaNodos; i++){
        printf("%c, ", lbl);
        lbl++;
    }
}
```

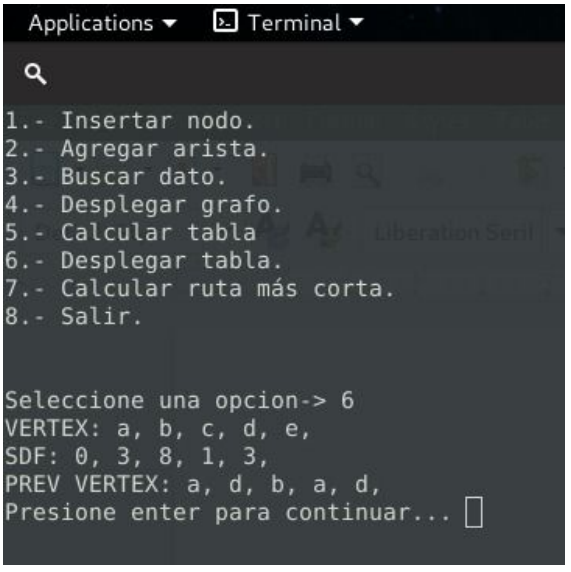
Ejecución del programa

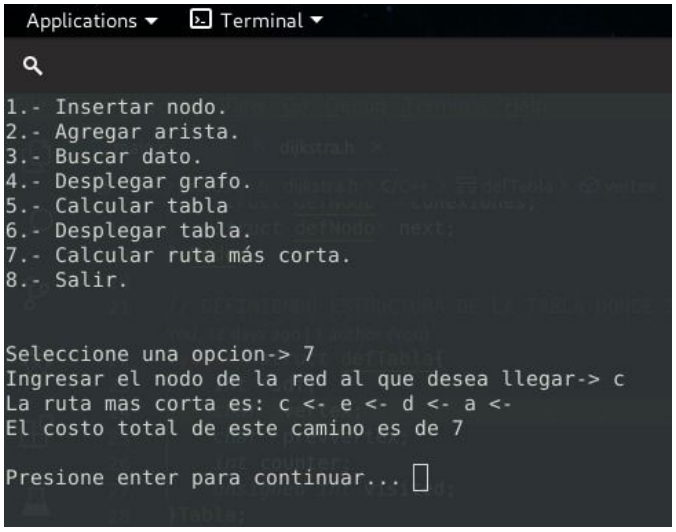

Applications
Terminal
1.- Insertar nodo.
2.- Agregar arista.
3.- Buscar dato.
4.- Desplegar grafo.
5.- Calcular tabla
6.- Desplegar tabla.
7.- Calcular ruta más corta.
8.- Salir.
Seleccione una opcion-> 2
Ingresar el nodo al que desea agregar la arista-> d
Ingresar el nodo al que se va a conectar el nodo anterior-> b
Ingresar costo del salto: 2
Se ha agregado la arista con exito
Presione enter para continuar...

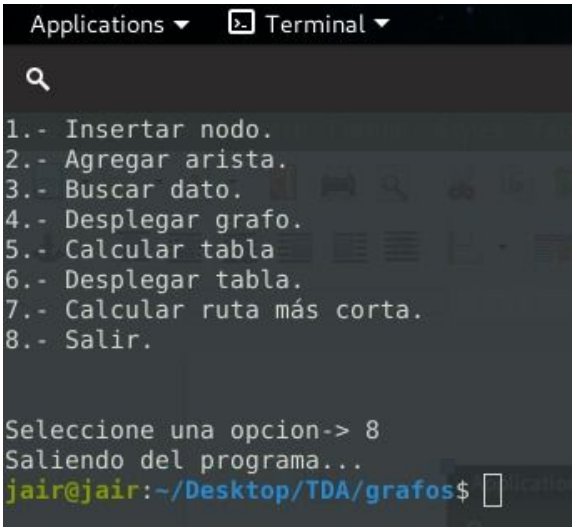

Applications
Terminal
1.- Insertar nodo.
2.- Agregar arista.
3.- Buscar dato.
4.- Desplegar grafo.
5.- Calcular tabla
6.- Desplegar tabla.
7.- Calcular ruta más corta.
8.- Salir.
Seleccione una opcion-> 3
Ingresar nodo a buscar-> b
El dato b ha sido encontrado en el grafo
Sus conexiones son:
Nodo: a, Costo: 6
Nodo: c, Costo: 5
Nodo: e, Costo: 2
Nodo: d, Costo: 2
Presione enter para continuar...


Applications
Terminal
1.- Insertar nodo.
2.- Agregar arista.
3.- Buscar dato.
4.- Desplegar grafo.
5.- Calcular tabla
6.- Desplegar tabla.
7.- Calcular ruta más corta.
8.- Salir.
Seleccione una opcion-> 4
a, b, c, d, e,
Presione enter para continuar...


Applications
Terminal
1.- Insertar nodo.
2.- Agregar arista.
3.- Buscar dato.
4.- Desplegar grafo.
5.- Calcular tabla
6.- Desplegar tabla.
7.- Calcular ruta más corta.
8.- Salir.
Seleccione una opcion-> 5
Tabla calculada
Presione enter para continuar...


Applications
Terminal
1.- Insertar nodo.
2.- Agregar arista.
3.- Buscar dato.
4.- Desplegar grafo.
5.- Calcular tabla
6.- Desplegar tabla.
7.- Calcular ruta más corta.
8.- Salir.
Seleccione una opcion-> 6
VERTEX: a, b, c, d, e,
SDF: 0, 3, 8, 1, 3,
PREV VERTEX: a, d, b, a, d,
Presione enter para continuar...





Conclusión:

Gracias a esta práctica comprendimos cómo funcionan los algoritmos de reconocimiento de rutas más óptimas que se utilizan en distintas aplicaciones comerciales, gracias a la implementación propia que hicimos de uno de estos (en este caso Dijkstra).

En esta práctica utilizamos conocimientos de temas de C desde los relativamente básicos (cómo Arreglos y ciclos) hasta los más complejos (como recursión y estructuras dinámicas) para cumplir las especificaciones dadas.

Además, para optimizar el proceso de codificación del programa, utilizamos herramientas que nosotros conocíamos previamente, como Git.