Tarea 2

Alejandro Arratia

Junio de 2024

Índice

1. Introducción **2**

2. Desarrollo **3**

2.1. Implementación del Grafo (DAG) 4

2.2. Herramientas de Sincronización Utilizadas para Establecer el Límite de Threads por Arista 6

2.3. Implementación del Registro Global 7

3. Resultados **8**

4. Análisis **9**

5. Conclusión **10**

1. Introducción

En este informe se presenta el desarrollo y análisis de un programa en C diseñado para encontrar la ruta de menor costo entre dos nodos dentro un grafo acíclico dirigido utilizando múltiples threads de ejecución. La concurrencia y la sincronización de hilos se manejan mediante semáforos y locks de mutex, lo cual asegura el acceso controlado a los recursos compartidos durante la ejecución del programa.

- **Descripción del Problema**
  Uno de los profesores de Sistemas Operativos está ansioso por encontrar una buena ruta para ir desde su casa hasta la Escuela de Ingeniería y Tecnología (EIT). Sin embargo, debido a sus múltiples responsabilidades, que incluyen revisar controles, preparar exámenes, escribir papers, y por supuesto disfrutar de juegos en su PC y Nintendo Switch, no dispone del tiempo suficiente para explorar manualmente todas las posibles rutas. Por lo tanto, ha decidido delegar esta tarea a un programa computacional.

- **Objetivos**

  1. Desarrollar un programa concurrente: Utilizar múltiples hilos para explorar el grafo y encontrar la ruta de menor costo.
  2. Implementar sincronización: Asegurar el acceso seguro y eficiente a los recursos compartidos mediante semáforos y mutexes.
  3. Maximizar el rendimiento: Optimizar el tiempo de ejecución permitiendo que los hilos trabajen en paralelo sin interferir entre sí.

- **Descripción general de la solución**

  1. Inicialización: Se configuran las estructuras de datos necesarias para representar el grafo y se inicializan las variables globales y los mecanismos de sincronización.
  2. Función de Búsqueda de Rutas: Cada hilo ejecuta una función que busca rutas desde el nodo inicial al nodo final, actualizando la ruta de menor costo encontrada globalmente.
  3. Gestión de Hilos: Se crean, ejecutan y sincronizan múltiples hilos para realizar la búsqueda de rutas. Una vez completada la búsqueda, se destruyen los recursos utilizados.
  4. Resultados: Al finalizar el tiempo de ejecución, se imprime la ruta de menor costo encontrada junto con el hilo que la descubrió.

A lo largo de este informe, se detallarán cada uno de estos componentes, proporcionando ejemplos de segmentos de código relevantes y explicaciones sobre su funcionamiento y propósito dentro del programa.

2. Desarrollo

El programa está diseñado para encontrar la ruta de menor costo en un grafo utilizando múltiples hilos de ejecución. En primer lugar, la función recibe como parámetro un puntero a una estructura Threads, la cual contiene información relevante para la ejecución del hilo, incluyendo el ID del hilo y un puntero al grafo sobre el cual se realizará la búsqueda.

Dentro de un bucle principal, la función verifica si ha transcurrido un minuto desde el inicio de la ejecución. Si este es el caso, se establece una variable de control para detener la ejecución de los hilos. Luego, se procede a inicializar variables necesarias para la búsqueda, como el nodo inicial, el nodo final y la estructura para almacenar la ruta actual que se está explorando.

A continuación, se inicia un bucle de búsqueda de ruta, donde se elige de forma aleatoria un nodo adyacente al nodo actual, avanzando hacia él y teniendo en cuenta el costo asociado a la arista que los conecta. Este proceso se repite hasta que se llega al nodo final o no hay más nodos adyacentes disponibles.

Si se encuentra una ruta completa (es decir, se llega al nodo final), se verifica si el costo total de esta ruta es menor que el costo mínimo global encontrado hasta el momento. En caso afirmativo, se actualiza la información sobre la mejor ruta encontrada hasta el momento. En caso de que no corresponda al nodo final y no hayan más rutas posibles, el nodo simplemente vuelve al inicio del bucle.

Finalmente, se liberan los recursos asignados dinámicamente para la ruta explorada y se pausa la ejecución del hilo por un breve período antes de reiniciar el bucle.

Al detenerse todos los hilos, se imprime la mejor ruta encontrada y su correspondiente costo mínimo global, proporcionando así el resultado final del proceso de búsqueda de ruta en el grafo dado.

Las herramientas utilizadas en este programa son:

- Librerías de C: pthread para la creación y manejo de hilos, semaphore para la sincronización mediante semáforos, y time para medir el tiempo de ejecución.
- Estructuras de Datos: Se define una estructura para representar las aristas del grafo (Edge) y otra para el grafo completo (Graph).
- Variables Globales y Sincronización: Se utilizan variables globales protegidas por mutexes para almacenar el costo mínimo global y la ruta correspondiente.

2.1. Implementación del Grafo (DAG)

El grafo debe ser dirigido acíclico, es decir, nosotros limitamos las conexiones que existen entre los nodos, y estos no pueden ser capaces de devolverse. Se crean las estructuras del Grafo en sí, que contiene los números de los nodos y las aristas junto con la estructura de las aristas (Edges), que contienen el nodo de origen, destino y el costo.

```
typedef struct {
    int source;
    int destination;
    int cost;
    sem_t semaphore;
} Edge;

typedef struct {
    int numNodes;
    int numEdges;
    Edge *edges;
} Graph;
```

Listing 1: Definición de la estructura del grafo

Las aristas se inicializan indicando su nodo de origen, nodo siguiente, costo y se le asigna un semáforo a cada una para controlar el acceso concurrente de threads.

```
Edge edges[] = {
        {0, 1, 1, {0}},
        {0, 2, 2, {0}},
        {1, 3, 3, {0}},
        {1, 4, 1, {0}},
        {2, 4, 2, {0}},
        {2, 5, 3, {0}},
        {3, 6, 1, {0}},
        {3, 7, 2, {0}},
        {4, 7, 3, {0}},
        {4, 8, 1, {0}},
        {5, 8, 2, {0}},
        {5, 9, 3, {0}},
        {6, 10, 1, {0}},
        {7, 10, 2, {0}},
        {7, 11, 3, {0}},
        {8, 11, 1, {0}},
        {8, 12, 2, {0}},
        {9, 12, 3, {0}},
        {9, 13, 1, {0}},
        {10, 14, 2, {0}},
        {11, 14, 3, {0}},
        {11, 15, 1, {0}},
        {12, 15, 2, {0}},
        {12, 16, 3, {0}},
        {13, 16, 1, {0}},
        {13, 17, 2, {0}},
```

```
28          {14, 18, 3, {0}},
29          {15, 18, 1, {0}},
30          {16, 18, 2, {0}},
31          {18, 19, 3, {0}}
32      };
```

Listing 2: Inicialización de aristas

En la siguiente figura podemos observar el grafo representado en un diagrama, con el número de los nodos indicado dentro de los cuadrados, en rojo los costos de cada ruta y la dirección asociada.

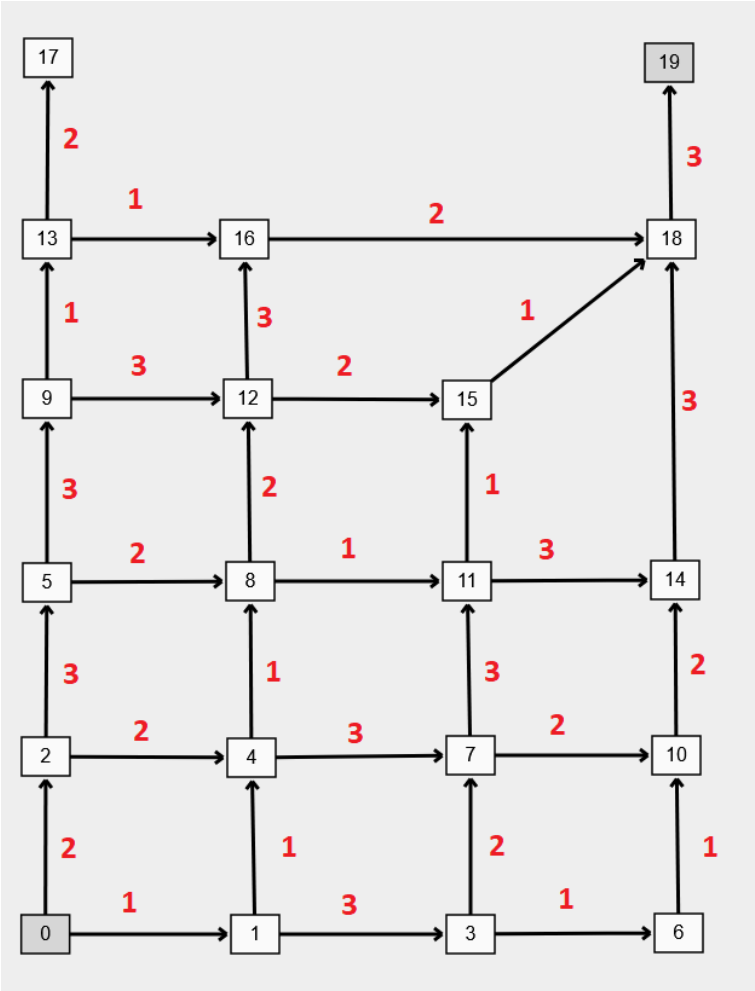

Figura 1: Diagrama del Grafo

Se puede ver la existencia de los 20 nodos mínimos del problema con el nodo 0 siendo el inicial y el nodo 19 el final. Además se ven las 30 aristas creadas anteriormente con sus respectivos costos y direcciones.

2.2. Herramientas de Sincronización Utilizadas para Establecer el Límite de Threads por Arista

Se comienza recibiendo por consola los valores de las variables N y M que corresponden a la cantidad de threads totales (con valores posibles de 5, 10 o 20) y los límites de accesos concurrentes a cada arista (con valores posibles de 2 o 3) respectivamente.

```
int num_threads;
    int semaphore_limit;

    do {
        printf("Ingrese el valor N correspondiente a la cantidad de
            Threads (5, 10, or 20): ");
        scanf("%d", &num_threads);
    } while (num_threads != 5 && num_threads != 10 && num_threads != 20);

    do {
        printf("Ingrese el valor M correspondiente al l mite de threads
            por arista (2 or 3): ");
        scanf("%d", &semaphore_limit);
    } while (semaphore_limit != 2 && semaphore_limit != 3);
```

Listing 3: Ingreso de parámetros por consola

Luego utilizando estos valores se crean los threads, para cada uno se le asocia un ID comenzando desde 1 y se envían a la función find route que es la que se encargará de encontrar la ruta con menor costo.

```
for (int i = 0; i < num_threads; i++) {
    threadInfos[i].id = i + 1;
    threadInfos[i].graph = &route;
    if (pthread_create(&threads[i], NULL, find_route, (void *)&threadInfos
        [i]) != 0) {
        perror("Error in thread creation");
        return 1;
    }
}
```

Listing 4: Creación Threads

Para controlar el acceso concurrente a las aristas del grafo, se utilizan semáforos. Cada arista tiene un semáforo asociado que limita la cantidad de hilos que pueden acceder a ella simultáneamente. Los semáforos se inicializan con el valor M proporcionado anteriormente (2 o 3).

```
for (int i = 0; i < num_edges; i++) {
    if (sem_init(&(route.edges[i].semaphore), 0, semaphore_limit) != 0) {
        perror("Error in semaphore initialization");
        return 1;
    }
}
```

Listing 5: Inicialización semáforos

Durante la ejecución de cada thread, se utiliza sem wait para decrementar el semáforo al acceder a una arista y sem post para incrementarlo al liberarla. Dentro del semáforo se revisa la cantidad de threads dentro de la aritsa desde el nodo actual hacia el siguiente.

```
int next_node_index = rand() % num_neighbors;
int next_node = neighbors[next_node_index];

sem_wait(&(graph->edges[next_edge_index].semaphore));
            total_cost += graph->edges[next_edge_index].cost;
            route[route_index++] = next_node;
            current_node = next_node;
            sem_post(&(graph->edges[next_edge_index].semaphore));
```

Listing 6: Uso de semáforo en cada arista

2.3. Implementación del Registro Global

El costo mínimo y la ruta correspondiente se almacenan en variables globales protegidas por mutexes para asegurar la consistencia de los datos y que sólo un thread pueda acceder a ella a la vez. Cuando un hilo encuentra una ruta con un costo menor al registrado globalmente, actualiza estas variables.

```
int global_min_cost = INT_MAX;
int global_min_route[MAX_ROUTE_LENGTH];
int global_min_route_length = 0;
int thread_with_min_cost = -1;
pthread_mutex_t min_cost_mutex;
```

Listing 7: Variables globlaes

El hilo que encuentra una ruta válida compara su costo con el costo global mínimo. Si es menor, actualiza las variables globales dentro de una sección crítica protegida por un mutex que corresponden al costo menor, la ruta completa correspondiente y el ID del thread.

```
pthread_mutex_lock(&min_cost_mutex);
if (total_cost < global_min_cost) {
    global_min_cost = total_cost;
    global_min_route_length = route_index;
    for (int i = 0; i < route_index; i++) {
        global_min_route[i] = route[i];
    }
    thread_with_min_cost = thread->id;
}
pthread_mutex_unlock(&min_cost_mutex);
```

Listing 8: Implementación mutex

3. Resultados

Al ejecutar el programa cada thread entra en un bucle buscando la mejor ruta posible, si llega al nodo final, compara el costo del camino recorrido con el valor mínimo actual y si este es menor, actualiza las variables globales y vuelve al inicio del bucle. En caso de llegar a un nodo que no tiene mas caminos posibles, los valores de la ruta y costo del nodo se liberan y vuelve al inicio del bucle.

En la siguiente figura podemos observar en azul el camino con menor costo del grafo calculado manualmente.

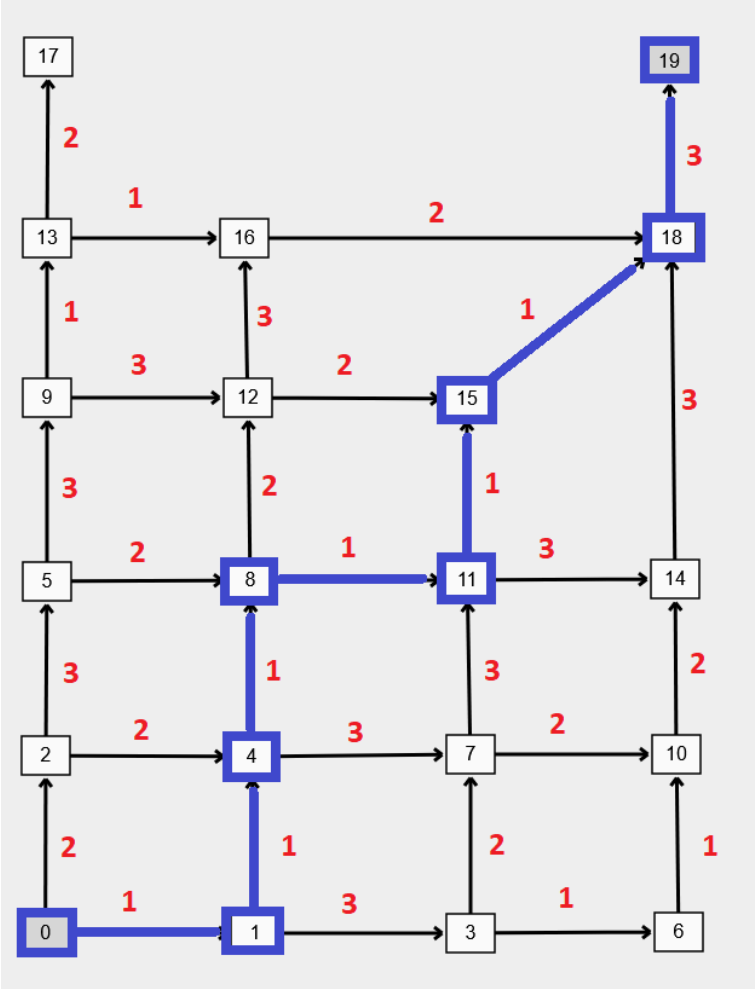

Figura 2: Ruta con menor costo

Podemos ver que el orden de la ruta es la siguiente: 0-1-4-8-11-15-18-19 y si sumamos los costos da un valor de 9.

Seguido de esto procedemos a ejecutar el programa con 20 threads y un límite de 3 hilos máximo por arista, y obtenemos lo siguiente por consola.

```
Ingrese el valor N correspondiente a la cantidad de Threads (5, 10, o 20): 20
Ingrese el valor M correspondiente al límite de threads por arista (2 o 3): 3
El programa se ejecutará por 60 segundos, utilizando 20 threads en total y 3 threads máximo por semáforo en cada Arista de los nodos
Thread 1 encontró un nuevo costo mínimo: 17, Con la ruta: 0 2 5 9 12 15 18 19
Thread 2 encontró un nuevo costo mínimo: 14, Con la ruta: 0 1 3 6 10 14 18 19
Thread 4 encontró un nuevo costo mínimo: 13, Con la ruta: 0 2 5 8 11 15 18 19
Thread 10 encontró un nuevo costo mínimo: 9, Con la ruta: 0 1 4 8 11 15 18 19
El tiempo ha finalizado
El Thread 10 encontró la ruta [0 1 4 8 11 15 18 19 ], que corresponde a la ruta con menor costo, con un valor de 9.
```

Figura 3: Respuesta en consola

Podemos ver que a se encuentran 4 rutas con costos diferentes y que cada ruta posee un costo menor que la anterior como era de esperarse con valores de 17, 14, 13 y 9 en orden de aparición. Los threads que encuentran cada ruta en esta ejecución en particular son diferentes.

El programa se ejecutó por 60 segundos, sin embargo encontró la mejor ruta en alrededor de 1 sólo segundo. Luego de que se acaba el tiempo de ejecución se imprime en consola que el Thread número 10 encontró la mejor ruta de costo 9 y que la misma que se calculó manualmente de 0-1-4-8-11-15-18-19.

4. Análisis

Pudimos obtener el resultado esperado, utilizando el programa para encontrar la ruta de menor costo dentro del grafo y contrastándola con el cálculo manual de ésta.

El uso de semáforos en las aristas del grafo nos permite controlar el acceso concurrente de los hilos a las mismas. Esta aplicación nos permite mantener un flujo controlado dentro del grafo lo que podría reducir condiciones de carrera o resultados inconsistentes. Sin embargo, el uso de semáforos puede introducir cierta sobrecarga debido a la necesidad de adquirir y liberarlos en cada iteración del bucle de búsqueda de ruta. Esta sobrecarga puede afectar la eficiencia del programa, especialmente en situaciones donde hay una gran cantidad de hilos compitiendo por el acceso a las aristas.

El programa además utiliza un mecanismo de mutex para proteger las variables globales compartidas que almacenan información sobre la ruta con el menor costo encontrado hasta el momento. Esto garantiza que solo un hilo pueda modificar estas variables a la vez, evitando condiciones de carrera y asegurando la consistencia de los resultados. La efectividad de este enfoque se refleja en la capacidad del programa para encontrar la ruta con el menor costo de manera precisa y sin corrupción de datos.

Un factor importante que se tuvo que modificar durante la implementación del código es que se tuvo que agregar una pausa minúscula al final de cada bucle, ya que sin esta pausa el primer thread creado alcanzaba a realizar varias iteraciones y encontrar la ruta menos

costosa antes de que siquiera se inicien los demas threads. Esto se puede deber al poder de procesamiento del computador y dado que el grafo utilizado como ejemplo es muy pequeño y no se requieren muchas iteraciones para encontrar la mejor ruta, en fin, esta micro pausa se utilizó para simular el comportamiento real en un grafo más complejo, en el que varios threads puedan participar.

5. Conclusión

El programa desarrollado en C cumple con los objetivos establecidos, utilizando múltiples hilos para explorar un grafo acíclico dirigido y encontrar la ruta de menor costo entre dos nodos. La implementación de la concurrencia y sincronización de hilos se realiza de manera efectiva mediante el uso de semáforos y locks de mutex, lo que garantiza un acceso controlado y seguro a los recursos compartidos durante la ejecución del programa.

El enfoque general de la solución sigue un proceso bien definido, comenzando con la inicialización de las estructuras de datos del grafo y la configuración de las variables globales y los mecanismos de sincronización. Posteriormente, cada hilo ejecuta una función dedicada a la búsqueda de rutas, actualizando la ruta de menor costo encontrada globalmente. La gestión de los hilos se realiza de manera eficiente, creándolos, ejecutándolos y sincronizándolos correctamente para llevar a cabo la búsqueda de rutas de manera concurrente. Finalmente, al completar el tiempo de ejecución, se presentan los resultados obtenidos, incluyendo la ruta de menor costo encontrada y el hilo que la descubrió, este resultado se comparó con la ruta obtenida manualmente y coincidían, verificando que el programa funcionaba correctamente.

La complejidad baja del grafo produce que no se encuentren muchas rutas diferentes posibles antes de encontrar la óptima de menor costo, causando que a pesar de que el programa se ejecuta sin parar durante 1 minuto en el que los hilos recorren miles de veces el grafo, la ruta correcta se encuentra en tan sólo 1 segundo aproximadamente. Este programa eso si, permite su escalabilidad, y si se utilizara un grafo de mayor complejidad y tamaño considerablemente mayor, se podría encontrar una gran cantidad de rutas posibles durante su ejecución.