Informe Tarea Nº2 - Sistemas Operativos

Martin Ignacio Aguilera Gomez - Felipe Solar Aracena
Martin.Aguilera@mail.udp.cl - Felipe.solar@mail.udp.cl

Integrantes:	Martin Aguilera
	Felipe Solar
Profesor:	Martin Guitierrez
Seccion:	01
Fecha:	06/02/2025

Índice de Contenidos

1. Introducion

En esta tarea se implementó un sistema de búsqueda de rutas en un grafo acíclico dirigido (DAG, por sus siglas en inglés) utilizando múltiples hilos (threads). El objetivo principal fue encontrar rutas óptimas desde un nodo inicial hasta un conjunto de nodos finales, analizando el impacto de la sincronización, la competencia por recursos y el acceso concurrente en la calidad de las soluciones obtenidas. Se desarrollaron dos versiones: una sin restricciones de acceso a los recursos compartidos (aristas del grafo), y otra con un límite sobre cuántos hilos pueden utilizar una misma arista de forma simultánea.

Este experimento busca simular situaciones reales en sistemas operativos donde múltiples procesos o hilos compiten por recursos limitados, y cómo el diseño del sistema afecta la eficiencia y los resultados de las operaciones concurrentes.

2. Desarrollo

2.0.1. Descripción del DAG y su función

Un **DAG** (Directed Acyclic Graph) es un grafo dirigido que no contiene ciclos. Esto significa que no es posible partir de un nodo, seguir una serie de aristas dirigidas y volver al mismo nodo. Esta propiedad es clave en contextos donde el orden importa y no puede haber retrocesos, como en la planificación de tareas, flujos de datos, compiladores, y flujos de decisiones.

En el contexto de este proyecto, el DAG representa las posibles rutas que puede tomar una tarea desde el nodo inicial (EIT) hasta los nodos finales. Cada nivel representa una etapa en el proceso, y cada arista (con un costo asociado) representa una transición posible entre dos nodos de niveles consecutivos.

2.1. Implementación del DAG

El grafo acíclico dirigido se construyó con 62 nodos distribuidos en 13 niveles. El nodo inicial es el nodo 0 (EIT), y cada nivel intermedio contiene 5 nodos. Las conexiones se generan desde cada nodo de un nivel hacia todos los nodos del siguiente nivel con costos aleatorios entre 5 y 20.

2.1.1. Código de generación del DAG

```
1  #include <iostream>
2  #include <vector>
3  #include <random>
4  #include <thread>
5  #include <mutex>
6  #include <chrono>
7  #include <fstream>
8  #include <climits>
9  #include <algorithm>
10 using namespace std;
11
```

```
12 struct Edge {
13     int vecino;
14     int costo;
15 };
16
17 class DAG_PATH_FINDER {
18 public:
19     vector<vector<Edge>> listaAdyacencia;
20     vector<vector<int>> niveles;
21     int mejoresCosto;
22     vector<int> mejoresRuta;
23     vector<pair<double, int>> HistorialCostos;
24     mutex mtx;
25     bool timeout;
26     chrono::steady_clock::time_point start;
27
28     DAG_PATH_FINDER() {
29         generarDAG();
30     }
31
32     void generarDAG() {
33         listaAdyacencia.resize(62);
34         niveles.resize(14);
35         niveles[0].push_back(0);
36
37         int nodo = 1;
38         for (int nivel = 1; nivel <= 12; ++nivel) {
39             for (int i = 0; i < 5; ++i) {
40                 niveles[nivel].push_back(nodo++);
41             }
42         }
43         niveles[13].push_back(61);
44         random_device rd;
45         mt19937 gen(rd());
46         uniform_int_distribution<> dist(5, 20);
47
48         for (int i = 0; i < 12; ++i) {
49             for (int u : niveles[i]) {
50                 for (int v : niveles[i + 1]) {
51                     int costo = dist(gen);
52                     listaAdyacencia[u].push_back({v, costo});
53                 }
54             }
55         }
56     }
57 };
```

2.1.2. Funcionamiento del código

La clase DAG_PATH_FINDER encapsula toda la lógica relacionada con el grafo. A continuación se detallan sus elementos más relevantes:

2.1.3. Funcionamiento del código

El siguiente fragmento de código implementa la generación del DAG explicado previamente. La clase DAG_PATH_FINDER encapsula toda la lógica relacionada con el grafo. A continuación se detallan sus elementos más relevantes:

- `listaAdyacencia`: Estructura que guarda para cada nodo sus conexiones salientes y el costo asociado a cada una (lista de adyacencia).
- `niveles`: Vector que agrupa los nodos según su nivel en el DAG (13 niveles en total).
- `generarDAG()`: Método que construye el DAG completo:
  1. Se inicializan los niveles y el nodo 0 se coloca como nodo inicial (nivel 0).
  2. Para cada uno de los 12 niveles siguientes, se crean 5 nodos y se asignan al nivel correspondiente.
  3. Se utiliza un generador aleatorio para asignar costos entre 5 y 20 a todas las conexiones posibles desde cada nodo del nivel actual hacia todos los nodos del nivel adyacente siguiente.
  4. El grafo es totalmente conexo entre niveles adyacentes, garantizando múltiples caminos posibles desde el inicio hasta los nodos finales.
- Variables como `mejoresCosto`, `mejoresRuta` y `HistorialCostos` serán usadas posteriormente para almacenar resultados durante la búsqueda de rutas óptimas.

Este diseño asegura que cualquier ruta válida comience en el nodo 0 y avance siempre hacia adelante en el grafo, sin posibilidad de ciclos, lo cual es esencial para aplicar algoritmos de búsqueda y optimización sin riesgo de loops infinitos.

2.1.4. Variables compartidas clave

- **mejoresCosto**: almacena el menor costo encontrado hasta el momento.
- **mejoresRuta**: secuencia de nodos correspondiente a la mejor ruta hallada.
- **timeout**: bandera para detener la ejecución tras cierto tiempo límite.
- **HistorialCostos**: pares de (tiempo, costo) que permiten graficar la evolución de la búsqueda.

2.2. Primera implementación: Sin límite

Esta versión implementa un enfoque simple pero efectivo: se lanzan **K hilos**, cada uno buscando rutas aleatorias desde el nodo inicial (EIT) hasta el nodo final (Casa, nodo 61), ubicado en el nivel 13 del DAG. No se imponen restricciones sobre cuántos hilos pueden explorar simultáneamente las mismas aristas, lo que permite un máximo aprovechamiento del paralelismo y una mayor velocidad de exploración.

Cada hilo ejecuta una función que construye caminos aleatorios hacia adelante en el DAG, acumulando el costo total. Si la ruta encontrada finaliza en el nodo 61 y tiene un costo menor que el mejor registrado hasta ese momento, se actualiza la mejor solución global.

2.2.1. Código de búsqueda aleatoria y ejecución paralela

```
1  void BuscarRutaAleatoria() {
2      random_device rd;
3      mt19937 gen(rd());
4
5      while (!timeout) {
6          vector<int> ruta;
7          int costoTotal = 0;
8          int actual = 0;
9          ruta.push_back(actual);
10
11         for (int nivel = 1; nivel <= 13; ++nivel) {
12             auto vecinos = listaAdyacencia[actual];
13             if (vecinos.empty()) break;
14
15             shuffle(vecinos.begin(), vecinos.end(), gen);
16             const Edge& e = vecinos.front();
17             costoTotal += e.costo;
18             actual = e.vecino;
19             ruta.push_back(actual);
20         }
21
22         if (actual == 61) {
23             lock_guard<mutex> lock(mtx);
24             if (costoTotal < mejoresCosto) {
25                 mejoresCosto = costoTotal;
26                 mejoresRuta = ruta;
27                 auto now = chrono::steady_clock::now();
28                 double segundos = chrono::duration<double>(now - start).count();
29                 HistorialCostos.push_back({segundos, costoTotal});
30                 cout << "Nuevo mejor costo encontrado: " << costoTotal << endl;
31                 ofstream out("actual.csv");
32                 for (int n : ruta) out << n << ",";
33                 out << "\n";
34             }
35         }
36     }
37 }
38
```

```
39 void EjecutarBusqueda(int K) {
40     mejoresCosto = INT_MAX;
41     mejoresRuta.clear();
42     timeout = false;
43     HistorialCostos.clear();
44     start = chrono::steady_clock::now();
45 
46     vector<thread> threads;
47     for (int i = 0; i < K; ++i) {
48         threads.emplace_back(&DAG_PATH_FINDER::BuscarRutaAleatoria, this);
49     }
50 
51     this_thread::sleep_for(chrono::seconds(60));
52     timeout = true;
53     for (auto& t : threads) t.join();
54 
55     lock_guard<mutex> lock(mtx);
56     cout << "\n== Mejor ruta encontrada (Costo: " << mejoresCosto << ") ==" << endl;
57     for (int n : mejoresRuta) cout << n << " ";
58     cout << endl;
59 
60     ofstream out("costos.csv");
61     out << "Tiempo,Costo\n";
62     for (auto& p : HistorialCostos)
63         out << p.first << "," << p.second << "\n";
64     out.close();
65 }
```

2.2.2. Función BuscarRutaAleatoria()

Esta función es ejecutada por cada hilo. Su lógica es la siguiente:

- Se inicializa un generador aleatorio de números.
- Mientras no se alcance el `timeout`, el hilo genera una ruta aleatoria:
  1. Comienza desde el nodo 0.
  2. Para cada uno de los 13 niveles siguientes (hasta llegar al nodo 61), elige aleatoriamente un vecino del nodo actual y lo sigue.
  3. Acumula el costo total del recorrido.
- Si la ruta termina exactamente en el nodo 61, se evalúa su costo. Si es mejor que el registrado previamente, se actualizan:
  - `mejoresCosto` con el nuevo valor.
  - `mejoresRuta` con la nueva ruta.
  - Se registra en `HistorialCostos` el tiempo transcurrido y el costo.
  - Se genera el archivo `actual.csv` con la mejor ruta encontrada hasta ese momento.

2.2.3. Función `EjecutarBusqueda(int K)`

Esta función orquesta la ejecución de múltiples hilos:

1. Inicializa las variables globales.
2. Crea `K` hilos, cada uno ejecutando `BuscarRutaAleatoria()`.
3. Espera 60 segundos mientras los hilos trabajan.
4. Señala el fin de la búsqueda con la variable `timeout`.
5. Espera que todos los hilos terminen con `join()`.
6. Muestra la mejor ruta encontrada por consola.
7. Genera un archivo `costos.csv` con la evolución del mejor costo en el tiempo.

2.2.4. Función principal

```
1 int main() {
2     DAG_PATH_FINDER buscador;
3     int K;
4     cout << "Ingrese el número de threads a usar (1, 10, 20, 50, 100): ";
5     cin >> K;
6     buscador.EjecutarBusqueda(K);
7     return 0;
8 }
```

En el `main()`, se crea un objeto de la clase `DAG_PATH_FINDER`, se solicita al usuario la cantidad de hilos a usar y se inicia la búsqueda. Esta implementación permite observar el impacto del paralelismo en la calidad de las soluciones encontradas y garantiza que las rutas alcancen correctamente el nodo final.

2.3. Segunda implementación: Con límite

En esta versión se introduce una restricción más realista al acceso compartido: cada arista solo puede ser utilizada simultáneamente por un número máximo **L** de hilos. Esto se logra mediante el uso de un `mutex` y un contador por cada arista.

Cuando un hilo desea utilizar una arista, primero adquiere el `mutex` correspondiente. Si el número actual de hilos usando esa arista es menor que `L`, se le permite continuar. Si no, libera el `mutex` y espera activamente (*busy-waiting*) antes de volver a intentar. Este mecanismo simula entornos donde los recursos compartidos tienen una capacidad limitada y no pueden ser accedidos indefinidamente por múltiples procesos al mismo tiempo.

2.3.1. Código de búsqueda aleatoria con límite por arista

```
void BuscarRutaAleatoria() {
    random_device rd;
    mt19937 gen(rd());

    while (!timeout) {
        vector<int> ruta;
        int costoTotal = 0;
        int actual = 0;
        ruta.push_back(actual);

        vector<pair<int, int>> aristasUsadas;

        for (int nivel = 1; nivel <= 13; ++nivel) {
            auto vecinos = listaAdyacencia[actual];
            if (vecinos.empty()) break;

            shuffle(vecinos.begin(), vecinos.end(), gen);
            const Edge& e = vecinos.front();
            pair<int, int> arco = {actual, e.vecino};

            while (true) {
                unique_lock<mutex> lock(*mtxAristas[arco]);
                if (usoAristas[arco] < L) {
                    usoAristas[arco]++;
                    break;
                }
                lock.unlock();
                this_thread::sleep_for(chrono::milliseconds(1));
            }

            aristasUsadas.push_back(arco);
            costoTotal += e.costo;
            actual = e.vecino;
            ruta.push_back(actual);
        }

        if (actual == 61) {
            lock_guard<mutex> lock(mtx);
            if (costoTotal < mejoresCosto) {
                mejoresCosto = costoTotal;
                mejoresRuta = ruta;
                auto now = chrono::steady_clock::now();
                double segundos = chrono::duration<double>(now - start).count();
                HistorialCostos.push_back({segundos, costoTotal});
                cout << "Nuevo mejor costo encontrado: " << costoTotal << endl;
                guardarRutaActualCSV(mejoresRuta);
                guardarHistorialCSV();
            }
        }

```

```
51          for (auto& arco : aristasUsadas) {
52              lock_guard<mutex> lock(*mtxAristas[arco]);
53              usoAristas[arco ]--;
54          }
55      }
56  }
```

2.3.2. Función BuscarRutaAleatoria()

Esta función ejecuta rutas aleatorias respetando el límite de uso concurrente por arista:

- Se inicializa un generador aleatorio.
- Mientras no se alcance el `timeout`, el hilo intenta construir una ruta desde el nodo 0 hasta el nodo 61.
- Para cada nivel:
  1. Se elige aleatoriamente una arista disponible desde el nodo actual.
  2. Antes de usarla, se adquiere un `mutex` y se verifica que el número de hilos en uso para esa arista sea menor que `L`.
  3. Si no hay disponibilidad, se espera activamente (`sleep_for`).
- Se acumula el costo y se almacena la ruta.
- Si la ruta termina en el nodo 61 y tiene mejor costo, se actualiza la mejor solución y se registran los datos.
- Finalmente, se libera el uso de cada arista empleada en la ruta.

2.3.3. Función principal

```
1  int main() {
2      int K;
3      cout << "Ingrese el número de threads a usar (1, 10, 20, 50, 100): ";
4      cin >> K;
5
6      int L;
7      cout << "Ingrese el límite de threads por arista (ej: 2 o 3): ";
8      cin >> L;
9
10     DAG_PATH_FINDER buscador(L);
11     buscador.start = chrono::steady_clock::now();
12
13     vector<thread> threads;
14     for (int i = 0; i < K; ++i)
15         threads.emplace_back(&DAG_PATH_FINDER::BuscarRutaAleatoria, &buscador);
16
17     this_thread::sleep_for(chrono::seconds(60));
18     buscador.timeout = true;
```

```
19
20      for (auto& t : threads)
21          t.join();
22
23      cout << "\n== Mejor ruta encontrada (Costo: " << buscador.mejoresCosto << ") ==\n";
24      for (int n : buscador.mejoresRuta)
25          cout << n << " ";
26      cout << endl;
27
28      return 0;
29  }
```

En el `main()`, se solicita al usuario tanto la cantidad de hilos `K` como el límite de concurrencia por arista `L`. Se inicializa el objeto `DAG_PATH_FINDER` con dicho límite, se lanzan los hilos y se ejecuta la búsqueda durante 60 segundos. Esta implementación permite analizar cómo la limitación del acceso concurrente a los recursos afecta la eficiencia en la búsqueda de rutas óptimas en un DAG.

3. Resultados experimentales y análisis

Se realizaron ejecuciones del algoritmo de búsqueda aleatoria en un DAG con 1, 10, 20, 50 y 100 hilos, bajo dos configuraciones:

- **Sin límite**: Las aristas pueden ser usadas por un número ilimitado de hilos simultáneamente.
- **Con límite (L=3)**: Cada arista puede ser usada por un máximo de 3 hilos a la vez.

A continuación se presenta un análisis comparativo de los resultados obtenidos.

3.1. Comparación de desempeño

Tiempo de convergencia

En las ejecuciones sin límite, los mejores costos se alcanzan rápidamente gracias a la alta concurrencia sin restricciones. En cambio, al imponer un límite, el sistema experimenta demoras adicionales debido a la contención en el uso de aristas, lo que retrasa la exploración de rutas más óptimas.

Por ejemplo:

- **Con 1 hilo**, ambos enfoques obtienen rutas óptimas en tiempos razonables, pero sin límite se alcanza un costo de 92 en 12.21 s, mientras que con límite se llega a 88 en 35.83 s.
- **Con 10 hilos**, sin límite se logra un costo de 93 en 24.18 s, mientras que con límite se obtiene 91 en 30.48 s.
- **Con 50 hilos**, la configuración sin límite alcanza un costo de 82 en menos de 1 s, mientras que con límite solo se llega a 95 en 22.86 s.

Calidad de las rutas encontradas

En general, las rutas con mejor costo se logran más fácilmente en la configuración sin límite. Esto se debe a que los hilos pueden explorar el DAG de forma más agresiva y sin bloqueos. En contraste, al imponer un límite por arista, se incrementa la contención, lo que restringe el espacio de búsqueda y hace más difícil encontrar rutas óptimas rápidamente.

- **Costo mínimo alcanzado (por cantidad de hilos):**
  - 1 hilo: **92 (sin límite)**, **88 (con límite)**
  - 10 hilos: **93 (sin límite)**, **91 (con límite)**
  - 20 hilos: **87 (sin límite)**, **89 (con límite)**
  - 50 hilos: **82 (sin límite)**, **95 (con límite)**
  - 100 hilos: **84 (sin límite)**, **87 (con límite)**

Efecto del límite por arista

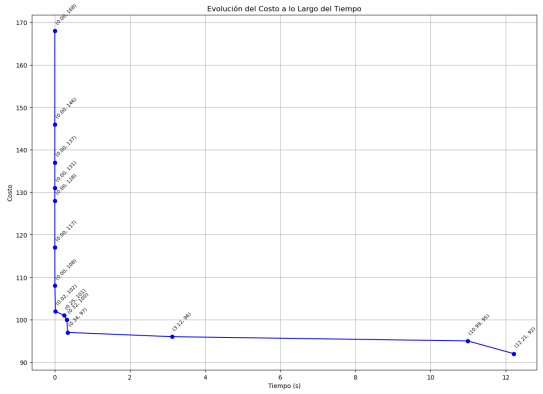
El límite por arista introduce un modelo más realista de acceso compartido a recursos, pero penaliza la escalabilidad. A medida que se incrementa el número de hilos, la configuración sin límite continúa encontrando mejores soluciones rápidamente, mientras que el modelo con límite sufre de mayor contención y estancamiento.

- A partir de 50 hilos, el impacto del límite es mucho más notorio: se alcanzan buenos costos sin límite en fracciones de segundo, mientras que con límite los hilos se bloquean frecuentemente esperando aristas disponibles.
- Con 100 hilos, la versión con límite no logra romper el umbral de costo 87, mientras que la versión sin límite llega a 84 en poco tiempo.

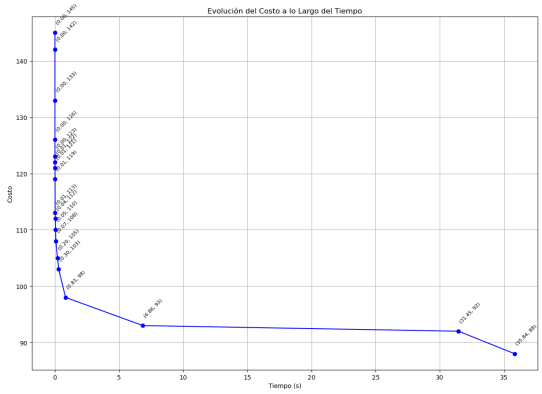
Resumen gráfico

A continuación se presentan las gráficas que muestran la evolución del costo a lo largo del tiempo para cada configuración y número de hilos. Estas permiten visualizar claramente:

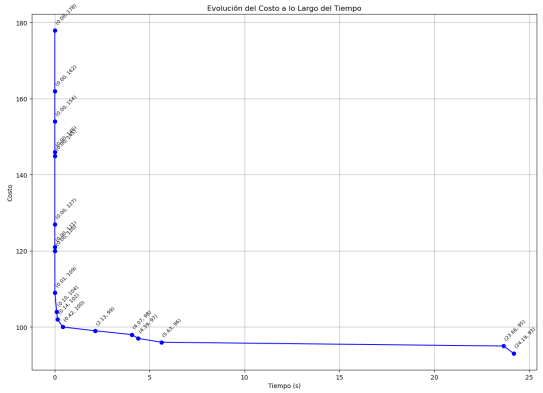
- La velocidad de convergencia de cada enfoque.
- La calidad de la mejor solución encontrada.
- El impacto del límite de concurrencia a medida que aumenta el paralelismo.



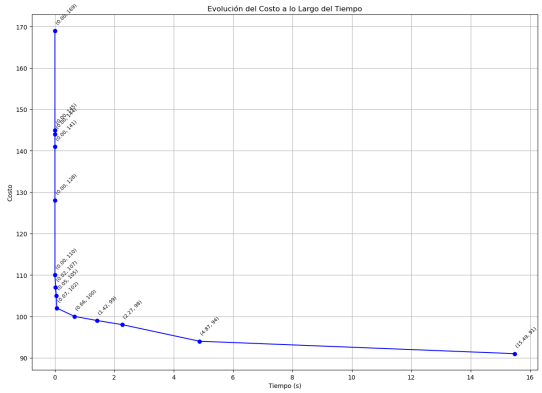
(a) Sin límite - 1 hilo



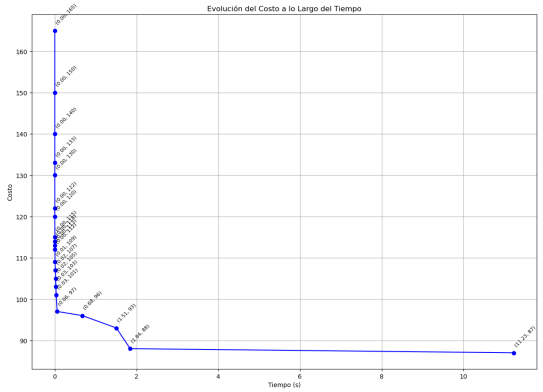
(b) Con límite $L = 3$ - 1 hilo



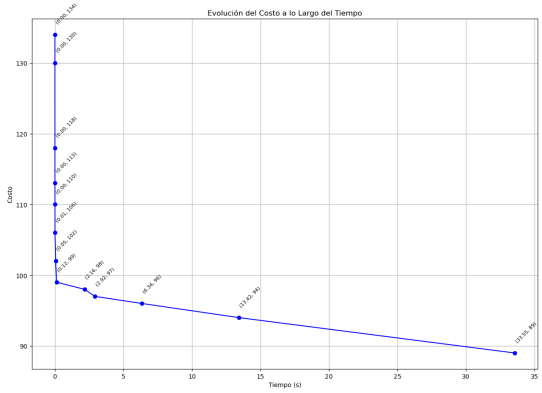
(c) Sin límite - 10 hilos



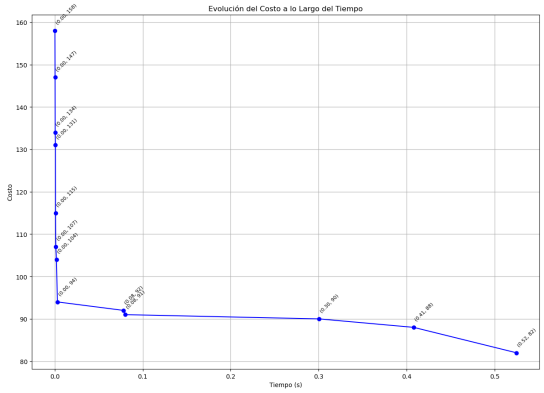
(d) Con límite $L = 3$ - 10 hilos



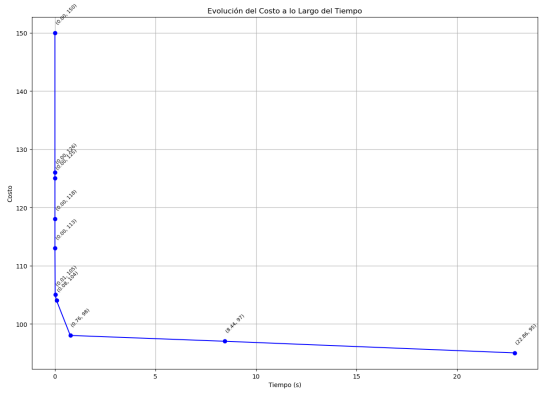
(e) Sin límite - 20 hilos



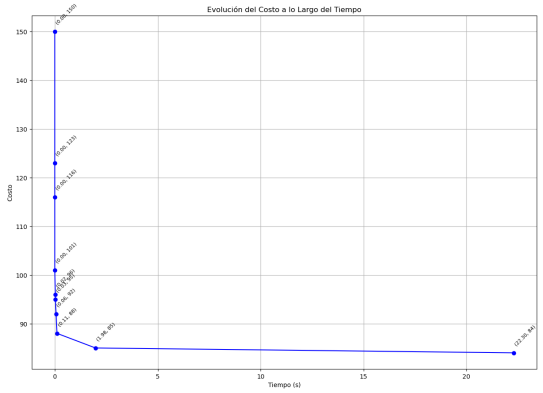
(f) Con límite $L = 3$ - 20 hilos



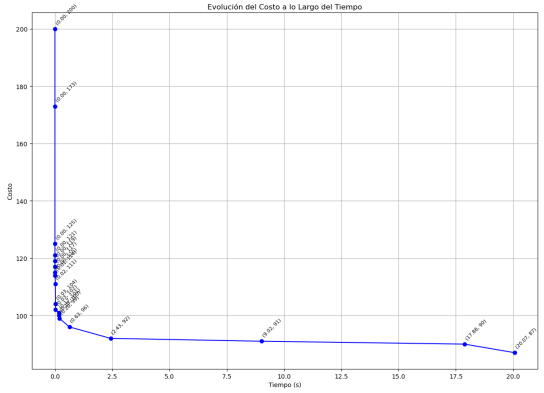
(a) Sin límite - 50 hilos



(b) Con límite $L = 3$ - 50 hilos



(c) Sin límite - 100 hilos



(d) Con límite $L = 3$ - 100 hilos

Figura 2: Evolución del costo con y sin límite de concurrencia por arista para distintas cantidades de hilos.

3.2. Conclusión

Los resultados experimentales muestran que la eliminación del límite de uso por arista favorece la exploración intensiva y rápida del DAG, especialmente cuando se utilizan muchos hilos. Sin embargo, este modelo puede ser irrealista en contextos donde los recursos compartidos tienen una capacidad limitada.
La introducción del límite $L = 3$ reduce la eficiencia, pero permite una simulación más cercana a escenarios reales, como redes de comunicación o sistemas de transporte, donde múltiples rutas no pueden ser usadas indefinidamente en paralelo.

4. Anexos

GitHub