UNIVERSIDAD NACIONAL DE SAN ANTONIO ABAD DEL CUSCO

FACULTAD DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA, INFORMÁTICA Y MECÁNICA

ESCUELA PROFESIONAL DE INGENIERÍA INFORMÁTICA Y DE SISTEMAS

Proyecto Final – Fase 1

Paralelización de Algoritmos Matriciales Masivos con CUDA: Warshall Lógico (Cerradura Transitiva Booleana)

Asignatura:

Algoritmos Paralelos y Distribuidos

Docente:

Mgt. Ray Dueñas Jiménez

Estudiantes:

Castro Pari, Rayneld Fidel
Mayhuire Chacon, Brenda Lucia
Mendoza Quispe, Jose Daniel
Perez Cahuana, Gabriel
Zevallos Yanqui, Andy Jefferson

Cusco - Perú
2025

Índice

Índice de figuras

Índice de cuadros

Resumen

La **cerradura transitiva booleana** (también llamada **Warshall lógico**) es un algoritmo clásico de teoría de grafos y álgebra booleana que determina la **alcanzabilidad** entre todos los pares de vértices de un grafo dirigido. Dada una matriz de adyacencia booleana $A \in \{0, 1\}^{N \times N}$, el algoritmo produce (in-place) una matriz T tal que $T_{ij} = 1$ si existe algún camino desde i hasta j. Su estructura es completamente matricial, con costo temporal $O(N^3)$ y costo espacial $O(N^2)$, lo cual lo convierte en un candidato fuerte para demostrar aceleración mediante GPU (CUDA) cuando N es masivo (requisito del proyecto: $N \geq 1024$).

En esta Fase 1 se presenta: (i) fundamento teórico y representación matricial; (ii) pseudocódigo y análisis de complejidad; (iii) una implementación secuencial en C optimizada a nivel básico (memoria contigua y atajo por A_{ik}); (iv) metodología de pruebas (reproducibilidad mediante semilla, densidad controlable p, y medición precisa del núcleo); y (v) el diseño conceptual para la Fase 2 (CUDA), incluyendo la paralelización 2D sobre (i, j) por cada iteración k, y consideraciones de coalescencia, sincronización y *tiling*.

Palabras clave: Cerradura transitiva, Warshall lógico, grafos, matrices booleanas, $O(N^3)$, CUDA, paralelización, rendimiento

Introducción

El cómputo moderno en ingeniería e informática trabaja con **matrices masivas** en múltiples contextos: conectividad de redes, dependencias entre módulos de software, grafos de estados, análisis de rutas, relaciones jerárquicas, entre otros. Cuando N crece (por ejemplo $N \geq 1024$), algoritmos cúbicos $O(N^3)$ se vuelven costosos en CPU, motivando el uso de GPU con CUDA.

En este marco, se propone el algoritmo de **Warshall lógico** para la Fase 1 del proyecto *Paralelización de Algoritmos Matriciales Masivos con CUDA*, debido a que:

- opera directamente sobre una matriz $N \times N$;
- tiene gran carga computacional ($O(N^3)$);
- su salida es verificable por comparación bit a bit;
- presenta paralelismo masivo en cada fase k sobre las celdas (i, j).

Objetivos

Objetivo general

Diseñar, implementar y evaluar (en Fase 2) la aceleración obtenida al paralelizar en CUDA el algoritmo de **cerradura transitiva booleana** (Warshall lógico) para matrices masivas ($N \geq 1024$), comparando contra una versión secuencial CPU.

Objetivos específicos

- Describir formalmente el problema de alcanzabilidad all-pairs y su modelado mediante matrices booleanas.
- Analizar complejidad temporal $O(N^3)$ y espacial $O(N^2)$, estimando magnitudes reales para $N \geq 1024$.
- Implementar una versión secuencial reproducible y medible, separando correctamente *núcleo* vs *overhead*.
- Establecer una metodología de pruebas: tamaños N, densidad p, semilla, validación para casos pequeños.
- Definir el plan de paralelización CUDA para Fase 2: mapeo de hilos, sincronización por k, y optimizaciones esperadas.

Marco teórico

Grafos dirigidos, matriz de adyacencia y alcanzabilidad

Un grafo dirigido $G = (V, E)$ con $|V| = N$ puede representarse mediante una matriz booleana de adyacencia A:

$$A_{ij} = \begin{cases} 1, & \text{si existe arco directo } i \to j \\ 0, & \text{caso contrario} \end{cases}$$

La **alcanzabilidad** (reachability) pregunta si existe un camino (de longitud ≥ 1) desde i hasta j. La **cerradura transitiva** T de A es una matriz tal que:

$$T_{ij} = 1 \iff \exists \text{ un camino de } i \text{ a } j.$$

Este problema aparece en bases de datos (consultas recursivas), análisis de dependencias y redes de comunicación.

Álgebra booleana en matrices

Warshall lógico aplica operaciones booleanas:

$$x \vee y \quad (\text{OR}), \qquad x \wedge y \quad (\text{AND}),$$

que en implementación pueden codificarse con enteros 0/1, o con operaciones bit a bit si se usan *bitsets*. La regla central es: "i llega a j si ya llegaba, o si llega a k y k llega a j".

Warshall lógico y relación con otros algoritmos

Warshall (1962) formaliza propiedades sobre matrices booleanas; su algoritmo computa cerradura transitiva. Floyd–Warshall, en cambio, computa distancias mínimas (con suma y mínimo) y comparte estructura triple-anidada. La versión booleana conserva el patrón cúbico, lo cual es útil como base didáctica para paralelización.

Descripción del algoritmo (Warshall lógico)

Nombre del algoritmo y campo de aplicación

El algoritmo estudiado se denomina **Warshall lógico** (o **cerradura transitiva booleana**). Pertenece al campo de **teoría de grafos** y **álgebra booleana**, y se aplica cuando se requiere resolver **alcanzabilidad all-pairs** (reachability) sobre un grafo dirigido representado matricialmente.

Campos de aplicación típicos:

- **Redes y enrutamiento**: determinar si un router/nodo puede comunicarse con otro a través de rutas multi-salto.

- **Análisis de dependencias en software**: saber si un módulo depende directa o indirectamente de otro (grafos de llamadas).
- **Bases de datos y consultas recursivas**: cierre transitivo en relaciones (por ejemplo, jerarquías o grafos de referencias).
- **Modelado de estados y verificación**: alcanzabilidad entre estados en autómatas/grafos de transición.
- **Procesamiento de imágenes (modelado como grafo)**: conectividad entre regiones/píxeles al representar adyacencias como grafo.

En todos estos casos, la salida T resume conectividad global y habilita consultas posteriores en tiempo $O(1)$ por par (i, j).

Recurrencia

Sea A la matriz booleana. El algoritmo actualiza:

$$A_{ij} \leftarrow A_{ij} \vee (A_{ik} \wedge A_{kj}), \qquad k = 0, \dots, N-1.$$

Interpretación: al fijar un k, se permite usar k como vértice intermedio; luego se incorpora a los intermedios permitidos.

Descripción detallada del funcionamiento (idea e invariante)

El algoritmo recorre un conjunto creciente de **vértices intermedios permitidos**. En la iteración k, se "habilita" el vértice k como posible intermedio en caminos de i a j.

Lectura operativa de la actualización:

$$A_{ij} \leftarrow A_{ij} \vee (A_{ik} \wedge A_{kj})$$

- A_{ij}: ya se conocía un camino (directo o descubierto previamente) de i a j.
- $(A_{ik} \wedge A_{kj})$: existe un camino de i a k y de k a j; por lo tanto existe un camino de i a j usando k como intermedio.
- El OR final acumula conocimiento: si era alcanzable antes o se vuelve alcanzable usando k, queda marcado como alcanzable.

Invariante (base formal de corrección). Sea $A^{(k)}$ la matriz luego de procesar el valor k (o equivalentemente, tras permitir intermedios en $\{0, \dots, k\}$). El invariante es:

$$A_{ij}^{(k)} = 1 \iff \exists \text{ un camino de } i \text{ a } j \text{ cuyos vértices intermedios están en } \{0, \dots, k\}.$$

Justificación breve:

- *Base*: para $k = -1$ (antes de iterar), $A^{(-1)}$ representa aristas directas (caminos sin intermedios).
- *Paso inductivo*: al pasar de $k-1$ a k, se mantiene lo ya alcanzable y se añade lo que se vuelve alcanzable usando al nuevo intermedio k (exactamente el término $A_{ik} \wedge A_{kj}$).

Al finalizar $k = N-1$, se han permitido todos los vértices como intermedios, obteniendo la cerradura transitiva completa.

Pseudocódigo

Algorithm 1 Cerradura transitiva booleana (Warshall lógico)

Require: Matriz booleana $A \in \{0,1\}^{N \times N}$
Ensure: A actualizada con su cerradura transitiva

```
for k = 0 to N − 1 do
    for i = 0 to N − 1 do
        for j = 0 to N − 1 do
            A[i][j] ← A[i][j] ∨ (A[i][k] ∧ A[k][j])
        end for
    end for
end for
```

Ejemplo controlado

Para $N = 4$ y:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

se representa la cadena $0 \to 1 \to 2 \to 3$, y la cerradura esperada es:

$$T = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Representación matricial y costos

Memoria requerida

Si se almacena cada celda como `uint8_t` (1 byte), la memoria para la matriz es N^2 bytes. La Tabla 1 resume órdenes típicos.

Cuadro 1: Memoria aproximada para A usando 1 byte por celda (sin contar overhead).

N	N^2	Memoria (bytes)	Aprox.
1024	1,048,576	1,048,576	$\sim$ 1 MB
2048	4,194,304	4,194,304	$\sim$ 4 MB
4096	16,777,216	16,777,216	$\sim$ 16 MB
8192	67,108,864	67,108,864	$\sim$ 64 MB

Representación matricial de entrada y salida

Entrada. La entrada es una matriz de adyacencia booleana $A \in \{0,1\}^{N\times N}$ donde cada celda codifica existencia de arco: $A_{ij} = 1$ si hay enlace directo $i \to j$, y $A_{ij} = 0$ en caso contrario.

Salida. La salida es la **cerradura transitiva** $T \in \{0,1\}^{N\times N}$ (en esta implementación se obtiene *in-place* sobre A), donde:

$$T_{ij} = 1 \iff \exists \text{ un camino (longitud } \geq 1) \text{ desde } i \text{ hasta } j.$$

Nota práctica: dependiendo de la convención, puede incluirse o no la alcanzabilidad reflexiva ($T_{ii} = 1$). En este documento se usa la convención de caminos de longitud ≥ 1 (por eso el ejemplo presenta diagonal cero).

Arreglo lineal en memoria (fila-major) para CPU/GPU

Para rendimiento y para facilitar copia a GPU, la matriz se almacena como un arreglo lineal contiguo (fila-major):

$$A_{ij} \longleftrightarrow A[i \cdot N + j].$$

Esta representación:

- reduce *overhead* de punteros (vs. `A[i][j]` con doble indirección),
- mejora localidad de caché en CPU,
- y en CUDA favorece accesos coalescentes cuando los hilos contiguos varían j.

Costo computacional $O(N^3)$ en magnitudes reales

El número de iteraciones del núcleo es N^3. Por ejemplo:

$$1024^3 = 1{,}073{,}741{,}824, \qquad 2048^3 = 8{,}589{,}934{,}592, \qquad 4096^3 = 68{,}719{,}476{,}736.$$

Esto explica por qué el algoritmo es ideal para evidenciar aceleración en GPU: la carga crece muy rápido con N.

Determinación y justificación formal de la complejidad (Big O)

El núcleo del algoritmo está compuesto por tres bucles anidados sobre k, i y j, cada uno recorriendo N elementos. En el **peor caso** (matriz densa), el cuerpo interno ejecuta un número constante de operaciones booleanas (lecturas, AND, OR y escritura), por lo que el costo total es:

$$T(N) = \sum_{k=0}^{N-1} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} O(1) = N \cdot N \cdot N \cdot O(1) = O(N^3).$$

Además, como el algoritmo necesariamente visita (en el peor caso) todas las combinaciones (k, i, j), también se cumple $T(N) = \Omega(N^3)$, y por tanto:

$$T(N) = \Theta(N^3).$$

Efecto del atajo por A_{ik}. La optimización "si $A_{ik} = 0$ continuar" puede reducir trabajo efectivo en matrices dispersas, pero **no cambia** la cota asintótica del peor caso: si la matriz es densa (muchos $A_{ik} = 1$), el algoritmo vuelve a ejecutar esencialmente los N^3 pasos, manteniendo $O(N^3)$.

Complejidad espacial. Se almacena una matriz booleana $N \times N$, por lo que el uso de memoria es:

$$S(N) = O(N^2).$$

Dependencias de datos

Existe una dependencia fuerte entre fases consecutivas k y $k+1$: todas las celdas (i, j) de la fase k deben completarse antes de pasar a $k+1$. Sin embargo, para un k fijo, las celdas (i, j) se actualizan de manera independiente, lo que habilita paralelismo masivo 2D.

Implementación secuencial (CPU) y mejoras básicas

Decisiones de implementación

Para medir correctamente el rendimiento y preparar la Fase 2:

- La matriz se almacena en un bloque contiguo de memoria (`uint8_t*`), facilitando accesos y futura copia a GPU.
- Se usa semilla fija para reproducibilidad, y densidad p para controlar lo disperso/denso del grafo.
- La medición cronometra **sólo el núcleo** (`k,i,j`), excluyendo generación e impresión.

Optimización local (atajo por A_{ik})

Una mejora simple y segura: si $A_{ik} = 0$, entonces $(A_{ik} \wedge A_{kj}) = 0$ para todo j, por lo que la fila i no cambia en esa fase k. Esto reduce trabajo cuando la matriz es dispersa.

Código C secuencial (medición y verificación para casos pequeños)

Listing 1: warshall_menu.c (CPU) con medición del nucleo y verificacion opcional

```c
// warshall_menu.c
// CPU secuencial: Cerradura transitiva booleana (Warshall logico)
// + MENU interactivo:
//   (1) ingresar matriz manual + parametros
//   (2) ingresar parametros + grafo random
//   (3) grafo y parametros random
//
// Mantiene modo clasico por argumentos:
//   ./warshall N p seed repeats verify [print]
//
// Compilar:
//   gcc -O3 -std=c11 warshall_menu.c -o warshall
// (si tu Linux requiere):
//   gcc -O3 -std=c11 warshall_menu.c -o warshall -lrt

#define _POSIX_C_SOURCE 200809L

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <ctype.h>

static inline double seconds_now(void) {
    struct timespec ts;
#if defined(CLOCK_MONOTONIC)
    if (clock_gettime(CLOCK_MONOTONIC, &ts) != 0) {
        perror("clock_gettime");
        exit(EXIT_FAILURE);
    }
#else
    if (clock_gettime(CLOCK_REALTIME, &ts) != 0) {
        perror("clock_gettime");
```

```c
        exit(EXIT_FAILURE);
    }
#endif
    return (double)ts.tv_sec + (double)ts.tv_nsec * 1e-9;
}

static uint8_t* alloc_matrix(int N) {
    size_t bytes = (size_t)N * (size_t)N * sizeof(uint8_t);
    uint8_t* A = (uint8_t*)malloc(bytes);
    if (!A) {
        fprintf(stderr, "ERROR: no se pudo asignar %zu bytes para N=%d\n", bytes, N
            );
        exit(EXIT_FAILURE);
    }
    return A;
}

static void init_random(uint8_t* A, int N, double p, unsigned seed) {
    srand(seed);
    for (int i = 0; i < N * N; i++) {
        double r = (double)rand() / (double)RAND_MAX;
        A[i] = (r < p) ? 1 : 0;
    }
}

static void print_matrix(const uint8_t* A, int N, const char* title) {
    printf("\n=== %s (N=%d) ===\n", title, N);

    printf("     ");
    for (int j = 0; j < N; j++) printf("%2d ", j);
    printf("\n");

    printf("     ");
    for (int j = 0; j < N; j++) printf("---");
    printf("\n");

    for (int i = 0; i < N; i++) {
        printf("%2d | ", i);
```

```
        const uint8_t* row = &A[i * N];
        for (int j = 0; j < N; j++) printf("%2d ", (int)row[j]);
        printf("\n");
    }
}

// -------------------------
// Nucleo: Warshall logico
// -------------------------
void warshall_logical(uint8_t* A, int N) {
    // A[i][j] = A[i][j] OR (A[i][k] AND A[k][j])
    // NO fuerza diagonal a 1.
    for (int k = 0; k < N; k++) {
        const uint8_t* row_k = &A[k * N];
        for (int i = 0; i < N; i++) {
            uint8_t aik = A[i * N + k];
            if (!aik) continue;
            uint8_t* row_i = &A[i * N];
            for (int j = 0; j < N; j++) {
                row_i[j] = (uint8_t)(row_i[j] | (aik & row_k[j]));
            }
        }
    }
}

// -----------------------------------
// Referencia: cerradura transitiva con BFS
// -----------------------------------
static void bfs_closure_ref(const uint8_t* Ain, uint8_t* Rref, int N) {
    int* queue = (int*)malloc((size_t)N * sizeof(int));
    uint8_t* vis = (uint8_t*)malloc((size_t)N * sizeof(uint8_t));
    if (!queue || !vis) {
        fprintf(stderr, "ERROR: memoria insuficiente para BFS\n");
        free(queue);
        free(vis);
        exit(EXIT_FAILURE);
    }

```

```
    for (int s = 0; s < N; s++) {
        memset(vis, 0, (size_t)N);
        int front = 0, back = 0;

        const uint8_t* row_s = &Ain[s * N];
        for (int v = 0; v < N; v++) {
            if (row_s[v]) {
                vis[v] = 1;
                queue[back++] = v;
            }
        }

        while (front < back) {
            int u = queue[front++];
            const uint8_t* row_u = &Ain[u * N];
            for (int v = 0; v < N; v++) {
                if (row_u[v] && !vis[v]) {
                    vis[v] = 1;
                    queue[back++] = v;
                }
            }
        }

        uint8_t* row_ref = &Rref[s * N];
        for (int j = 0; j < N; j++) row_ref[j] = vis[j];
    }

    free(queue);
    free(vis);
}

static int verify_against_ref(const uint8_t* Rref, const uint8_t* Aout, int N) {
    for (int i = 0; i < N; i++) {
        const uint8_t* rr = &Rref[i * N];
        const uint8_t* ao = &Aout[i * N];
        for (int j = 0; j < N; j++) {
            if (rr[j] != ao[j]) {
                fprintf(stderr,
```

```
                        "FALLO verificacion: fila i=%d, col j=%d | esperado=%d, obtenido=%d\n",
                        i, j, (int)rr[j], (int)ao[j]);
                return 0;
            }
        }
    }
    return 1;
}

// ====================================================
// Helpers de input robusto (sin scanf)
// ====================================================
static void read_line(char* buf, size_t n) {
    if (!fgets(buf, (int)n, stdin)) {
        printf("\nEOF detectado. Saliendo.\n");
        exit(0);
    }
}

static long read_long_prompt(const char* prompt, long minv, long maxv) {
    char line[256];
    for (;;) {
        printf("%s", prompt);
        read_line(line, sizeof(line));

        char* end = NULL;
        errno = 0;
        long v = strtol(line, &end, 10);
        if (errno == 0) {
            while (end && *end && isspace((unsigned char)*end)) end++;
            if (end && (*end == '\0' || *end == '\n')) {
                if (v >= minv && v <= maxv) return v;
            }
        }
        printf("Entrada invalida. Rango permitido: [%ld..%ld]\n", minv, maxv);
    }
}
```

```c

static double read_double_prompt(const char* prompt, double minv, double maxv) {
    char line[256];
    for (;;) {
        printf("%s", prompt);
        read_line(line, sizeof(line));

        char* end = NULL;
        errno = 0;
        double v = strtod(line, &end);
        if (errno == 0) {
            while (end && *end && isspace((unsigned char)*end)) end++;
            if (end && (*end == '\0' || *end == '\n')) {
                if (v >= minv && v <= maxv) return v;
            }
        }
        printf("Entrada invalida. Rango permitido: [%.3f..%.3f]\n", minv, maxv);
    }
}

static int read_int_prompt(const char* prompt, int minv, int maxv) {
    return (int)read_long_prompt(prompt, (long)minv, (long)maxv);
}

static int read_yesno_prompt(const char* prompt) {
    char line[64];
    for (;;) {
        printf("%s (1=si, 0=no): ", prompt);
        read_line(line, sizeof(line));
        if (line[0] == '1') return 1;
        if (line[0] == '0') return 0;
        printf("Entrada invalida. Escribe 1 o 0.\n");
    }
}

static int parse_row_01(const char* line, uint8_t* row, int N) {
    // Acepta: "0 1 0 1" o "0101..." (con o sin espacios)
    int count = 0;
```

```
    for (const char* p = line; *p && count < N; p++) {
        if (*p == '0' || *p == '1') {
            row[count++] = (uint8_t)(*p - '0');
        }
    }
    return (count == N);
}

static double density_ones(const uint8_t* A, int N) {
    long long ones = 0;
    for (long long i = 0; i < (long long)N * (long long)N; i++) ones += A[i] ? 1 :
        0;
    return (double)ones / (double)((long long)N * (long long)N);
}

// ====================================================
// Ejecutar en modo verify/timing (reusa tu logica)
// ====================================================
static int run_experiment(uint8_t* Ain, int N, double p, unsigned seed, int repeats
    , int verify, int print) {
    const int PRINT_LIMIT = 16;

    if (N <= PRINT_LIMIT) { // comportamiento original
        verify = 1;
        print = 1;
    }
    if (repeats <= 0) repeats = 1;

    uint8_t* A = alloc_matrix(N);

    if (verify) {
        if (N > 128) {
            printf("Aviso: verificacion activada con N=%d; se recomienda N<=128.\n"
                , N);
        }

        memcpy(A, Ain, (size_t)N * (size_t)N);

```

```c
        double t0 = seconds_now();
        warshall_logical(A, N);
        double t1 = seconds_now();
        double kernel_time = t1 - t0;

        uint8_t* Rref = alloc_matrix(N);
        bfs_closure_ref(Ain, Rref, N);

        int ok = verify_against_ref(Rref, A, N);

        if (print) {
            print_matrix(Ain,  N, "MATRIZ DE ENTRADA (Grafo / Adyacencia)");
            print_matrix(Rref, N, "MATRIZ DE VERIFICACION (Referencia BFS)");
            print_matrix(A,    N, "MATRIZ DE SALIDA (Warshall logico)");
        }

        printf("\nVALIDACION (BFS) para N=%d: %s\n", N, ok ? "OK" : "FALLIDA");
        printf("Tiempo del nucleo (warshall_logical): %.6f s\n", kernel_time);
        printf("Resumen params | N=%d | p=%.3f | seed=%u | repeats=%d | verify=%d |
            print=%d\n",
               N, p, seed, repeats, verify, print);

        free(Rref);
        free(A);
        return ok ? EXIT_SUCCESS : EXIT_FAILURE;
    }

    double best = 1e100;
    for (int r = 0; r < repeats; r++) {
        memcpy(A, Ain, (size_t)N * (size_t)N);
        double t0 = seconds_now();
        warshall_logical(A, N);
        double t1 = seconds_now();
        double dt = t1 - t0;
        if (dt < best) best = dt;
    }

```

```
    printf("CPU Warshall logico | N=%d | p=%.3f | seed=%u | repeats=%d | 
        best_kernel_time=%.6f s\n",
            N, p, seed, repeats, best);

    free(A);
    return EXIT_SUCCESS;
}

// ======================================================
// Menu
// ======================================================
static void menu_loop(void) {
    for (;;) {
        printf("\n==============================\n");
        printf("   MENU - Warshall logico CPU\n");
        printf("==============================\n");
        printf("1) Ingresar MATRIZ manual + parametros\n");
        printf("2) Ingresar parametros + GRAFO random\n");
        printf("3) Grafo y parametros RANDOM\n");
        printf("0) Salir\n");

        int opt = read_int_prompt("Opcion: ", 0, 3);
        if (opt == 0) break;

        int N = 256;
        double p = 0.05;
        unsigned seed = 1234;
        int repeats = 3;
        int verify = 0;
        int print = 0;

        uint8_t* Ain = NULL;

        if (opt == 1) {
            N = read_int_prompt("Ingrese N (1..2048 recomendado): ", 1, 4096);
            Ain = alloc_matrix(N);

            printf("\nIngrese la matriz de adyacencia (%dx%d) con 0/1.\n", N, N);
```

```
            printf("Formato permitido por fila: '0 1 0 1' o '0101...'\n\n");

            char line[8192];
            for (int i = 0; i < N; i++) {
                for (;;) {
                    printf("Fila %d: ", i);
                    read_line(line, sizeof(line));
                    if (parse_row_01(line, &Ain[i * N], N)) break;
                    printf("Fila invalida. Debe contener %d valores 0/1.\n", N);
                }
            }

            // p se calcula como densidad real (informativo)
            p = density_ones(Ain, N);
            seed = 0;

            repeats = read_int_prompt("repeats (>=1): ", 1, 1000);
            verify  = read_yesno_prompt("verify");
            print   = read_yesno_prompt("print");

            printf("\nDensidad p calculada desde la matriz: %.3f\n", p);
        }
        else if (opt == 2) {
            N = read_int_prompt("N (1..4096): ", 1, 4096);
            p = read_double_prompt("p (0..1): ", 0.0, 1.0);
            seed = (unsigned)read_long_prompt("seed (0..2^31-1): ", 0, 2147483647L)
                ;
            repeats = read_int_prompt("repeats (>=1): ", 1, 1000);
            verify  = read_yesno_prompt("verify");
            print   = read_yesno_prompt("print");

            Ain = alloc_matrix(N);
            init_random(Ain, N, p, seed);
        }
        else if (opt == 3) {
            // Random razonable (puedes ajustar los rangos)
            unsigned s = (unsigned)time(NULL);
            srand(s);
```

```c

            int choices[] = {8,16,32,64,128,256,512,1024};
            int idx = rand() % (int)(sizeof(choices)/sizeof(choices[0]));
            N = choices[idx];

            p = 0.01 + ((double)rand() / (double)RAND_MAX) * 0.19; // [0.01..0.20]
            seed = (unsigned)rand();
            repeats = 1 + (rand() % 7); // 1..7

            // si N tiene un size chico, validamos; si no, solo timing
            verify = (N <= 128) ? 1 : 0;
            print  = (N <= 16) ? 1 : 0;

            Ain = alloc_matrix(N);
            init_random(Ain, N, p, seed);

            printf("\nParametros random generados:\n");
            printf("N=%d | p=%.3f | seed=%u | repeats=%d | verify=%d | print=%d\n",
                   N, p, seed, repeats, verify, print);
        }

        int rc = run_experiment(Ain, N, p, seed, repeats, verify, print);
        free(Ain);

        if (rc != EXIT_SUCCESS) {
            printf("Ejecucion termino con error (validacion fallida o problema).\n"
                );
        }

        if (!read_yesno_prompt("\nDeseas ejecutar otra vez?")) break;
    }
}

int main(int argc, char** argv) {
    // --- Modo por argumentos (tu modo original) ---
    if (argc >= 2) {
        int N = 256;
        double p = 0.05;
```

```c
        unsigned seed = 1234;
        int repeats = 3;
        int verify = 0;
        int print = 0;

        if (argc >= 2) N = atoi(argv[1]);
        if (argc >= 3) p = atof(argv[2]);
        if (argc >= 4) seed = (unsigned)atoi(argv[3]);
        if (argc >= 5) repeats = atoi(argv[4]);
        if (argc >= 6) verify = atoi(argv[5]);
        if (argc >= 7) print = atoi(argv[6]);

        if (N <= 0) { fprintf(stderr, "ERROR: N debe ser > 0\n"); return EXIT_FAILURE; }
        if (p < 0.0 || p > 1.0) { fprintf(stderr, "ERROR: p debe estar en [0,1]\n"); return EXIT_FAILURE; }
        if (repeats <= 0) repeats = 1;

        const int PRINT_LIMIT = 16;
        if (N <= PRINT_LIMIT) { verify = 1; print = 1; }

        uint8_t* Ain = alloc_matrix(N);
        init_random(Ain, N, p, seed);

        int rc = run_experiment(Ain, N, p, seed, repeats, verify, print);
        free(Ain);
        return rc;
    }

    // --- Modo menu ---
    menu_loop();
    return 0;
}
```

Como se puede apreciar, el código incluye una función de verificación, que comprueba la salida de nuestro código usando BFS (búsqueda en anchura) para determinar si hay o no camino los vértices que se indican en la salida de nuestro algoritmo.

Metodología de pruebas y análisis experimental

Variables controladas

Para asegurar comparabilidad:

- **Semilla fija**: garantiza mismos datos en ejecuciones repetidas.
- **Densidad** p: controla la probabilidad de 1 (matriz dispersa vs densa).
- **Medición del núcleo**: sólo el tiempo del triple bucle (excluye generación e I/O).

Tamaños de prueba

Se recomienda ejecutar:

$$N \in \{128, 256, 512, 1024, 2048\}$$

(extendible a 4096 si la memoria/tiempo lo permite). Para esta fase, lo crítico es incluir $N \geq 1024$.

Compilación y ejecución

```
# Compilar (Linux)
gcc -O3 -std=c11 warshall_menu.c -o warshall

# Ejecutar: ./warshall N p seed repeats verify print
./warshall 256  0.05 1234 5 0 0
./warshall 1024 0.02 1234 3 0 0
./warshall 2048 0.02 1234 3 0 0
./warshall 4096 0.005 1234 2 0 0
```

Plantilla de resultados (CPU secuencial)

Cuadro 2: Resultados de tiempo (CPU secuencial).

N	N^2	N^3	p	Tiempo núcleo (s)
128	16,384	2,097,152	0.05	0.000062
256	65,536	16,777,216	0.05	0.000808
512	262,144	134,217,728	0.02	0.008243
1024	1,048,576	1,073,741,824	0.02	0.075686
2048	4,194,304	8,589,934,592	0.02	0.630069
4096	16,777,216	6,719,476,736	0.02	5.269098

En la siguiente imagen se evidencia el crecimiento cúbico del tiempo con respecto a N.

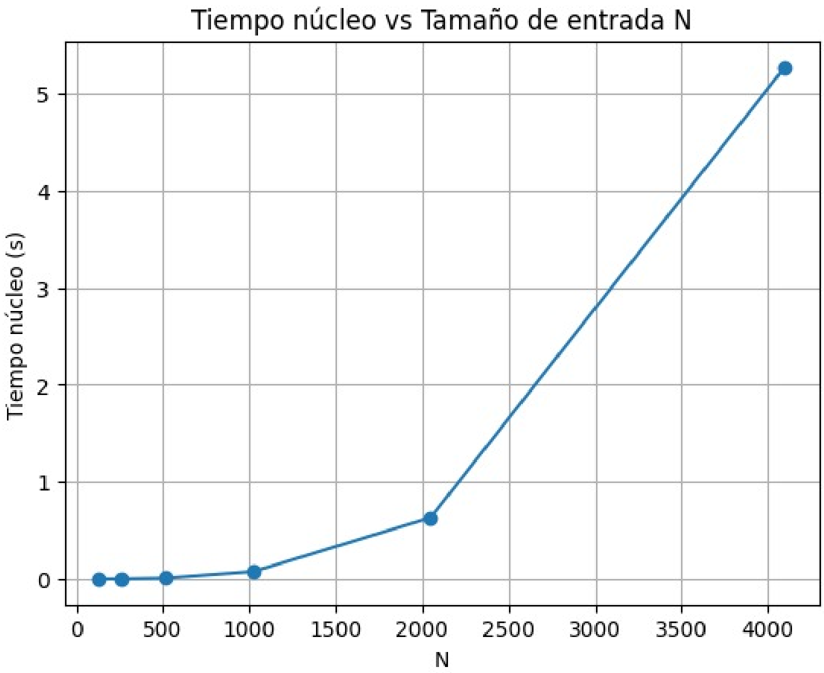

Figura 1: Tiempo del núcleo vs. N (CPU secuencial).

Discusión de resultados

- Al duplicar N, el tiempo tiende a crecer aproximadamente por un factor cercano a 8 (por N^3).
- Para matrices dispersas (p pequeño), el atajo por A_{ik} puede reducir significativamente el trabajo efectivo.
- Para matrices densas (p grande), la mejora por el atajo disminuye y el costo se acerca más al cúbico completo.

Conclusiones

- Warshall lógico computa cerradura transitiva booleana sobre una matriz $N \times N$ y resuelve alcanzabilidad all-pairs.
- Su costo temporal $O(N^3)$ y su estructura matricial lo convierten en un candidato fuerte para aceleración con CUDA cuando $N \geq 1024$.
- La dependencia por fases en k obliga a sincronizar entre iteraciones, pero dentro de cada fase existe paralelismo masivo 2D.
- La implementación secuencial propuesta es reproducible y medible (semilla fija, densidad p, medición del núcleo), permitiendo comparar con la futura versión CUDA de manera confiable.
- Para Fase 2, el diseño recomendado es un kernel por k con grilla 2D, cuidando coalescencia y (opcionalmente) *tiling* para reducir accesos a memoria global.