Resolución de Tarea 9 - Algoritmos Probabilisticos y Aproximados (Fecha: 8 de Diciembre de 2025)

Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI5651 - Diseño de Algoritmos I
Septiembre - Diciembre 2025
Estudiante: Junior Miguel Lara Torres (17-10303)

Tarea 9 (9 puntos)

Indice

Pregunta 1

El carnet en cuestión es 1710303, por lo tanto se tiene

$$N = 171030317103031710303$$

Se usó Python, el archivo se encuentra en miller_rabin_rep.py donde se obtuvo

- Iteración 1: base 'a' = 143110979125382574214
  - Valor de t: 90546959054695905469
  - Valor de s: 1
  - Descomposición N-1: $2^s \cdot t \to 2^1 \cdot 90546959054695905469$
  - Valor de x: $x = a^t \ mod \ n \to 143110979125382574214^{90546959054695905469} \ mod \ 181093918109391810939 = 99839335598454313666$

Apreciando el resultado tenemos que como $x \neq 1 \wedge x \neq N-1 \wedge s = 1$, entonces BTest retorna falso. Determinando así, que N no es un número primo.

En este sentido, no fue necesario ejecutar las 10 iteraciones establecidas dada la condición de BTest $x \neq 1 \wedge x \neq N-1$, en su defecto bastó con la primera iteración.

Pregunta 2

Se solicita diseñar e implementar un algoritmo probabilístico de Monte Carlo para verificar si una matriz B es la inversa de una matriz A (es decir, $A \times B = I$, donde I es la matriz identidad), con una probabilidad de error acotada por ϵ.

Utilizaremos el **Algoritmo de Freivalds** adaptado. El test se basa en el principio de que, si $A \times B \neq I$, es muy probable que una multiplicación matricial-vectorial aleatoria revele la desigualdad.

- Verificar $A \times B = I$ es equivalente a verificar si la matriz de diferencia $D = A \times B - I$ es la matriz nula.

- En lugar de calcular D (lo cual es $O(n^3)$), seleccionamos un vector aleatorio X de tamaño $1 \times n$ (donde $X_i \in \{0, 1\}$). Comprobamos si $X \cdot D = \mathbf{0}$, o lo que es lo mismo: $X \cdot A \cdot B = X \cdot I$. Dado que $X \cdot I = X$, la prueba se reduce a:
- $$X \cdot (A \times B) = X$$
- La multiplicación $X \cdot A$ (vector fila por matriz) toma $O(n^2)$. Luego, $(X \cdot A) \cdot B$ (vector fila por matriz) toma otros $O(n^2)$. El test se ejecuta en $\mathbf{O(n^2)}$ por iteración.
- Si $A \times B \neq I$, la probabilidad de que la prueba falle (produzca **X** en la salida) es a lo sumo $1/2$. Para asegurar que la probabilidad de error sea menor que ϵ, debemos repetir el test k veces, donde k se define como $k = \lceil \log_2(1/\epsilon) \rceil$.

Así, el algoritmo implementado (CheckInversion) verifica la relación $A \times B = I$ utilizando la prueba de Freivalds repetida k veces. Sabiendo que multiplicando el costo por iteración por el número de iteraciones la complejidad total queda como:

$$O(n^2 \log(1/\epsilon))$$

Implementación en C++

El archivo funcional se encuentra en check_inversion_matrix.cpp

```cpp
typedef long long MatElement;
typedef vector<vector<MatElement>> Matrix;
typedef vector<MatElement> Vector;

// Generador de números aleatorios
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());

/**
 * Genera un vector aleatorio X (1 x N) con elementos {0, 1}.
 */
Vector GenerateRandomVector(int N)
{
    uniform_int_distribution<MatElement> dist(0, 1);
    Vector X(N);
    for (int i = 0; i < N; ++i)
        X[i] = dist(rng);
    return X;
}

/**
 * Calcula el producto de un vector fila X por una matriz A: R = X * A.
 * R y X son vectores 1D de tamaño N.
 * Complejidad: O(N^2).
 */
Vector MatrixVectorProduct(const Vector &X, const Matrix &A, int N) {
    Vector R(N, 0);
    // Multiplicación X (1xN) * A (NxN) = R (1xN)
    for (int j = 0; j < N; ++j) { // columna de A / elemento de R
        for (int i = 0; i < N; ++i)
            R[j] += X[i] * A[i][j]; // fila de X / fila de A
    }
    return R;
}

/**
 * Algoritmo de Monte Carlo para verificar si B = A^-1 (A * B = I)
 * Retorna true si probablemente es la inversa, false si es definitivamente compuesta.
 */
```

```cpp
bool CheckInversion(const Matrix &A, const Matrix &B, int N, double epsilon) {
    if (N == 0)
        return true;

    // Calcular K: número de iteraciones necesarias para error < epsilon
    // K = ceil(log2(1/epsilon))
    int K = max(1, (int)ceil(log2(1.0 / epsilon)));

    for (int k = 1; k <= K; ++k) {
        // 1. Generar vector aleatorio X
        Vector X = GenerateRandomVector(N);

        // 2. Calcular R_AB = (X * A) * B
        Vector R_A = MatrixVectorProduct(X, A, N);
        Vector R_AB = MatrixVectorProduct(R_A, B, N);

        // 3. Comparar R_AB con X (deberían ser iguales si A*B = I)
        bool match = true;
        for (int i = 0; i < N; ++i) {
            if (R_AB[i] != X[i]) {
                match = false;
                break;
            }
        }

        cout << "Iteracion " << k << "/" << K << ": Resultado: " << (match ? "COINCIDE (Probab

        if (!match) {
            cout << "\nCONCLUSION: B NO es la inversa de A (testigo encontrado)." << en
            return false;
        }
    }

    cout << "\nCONCLUSION: B es probablemente la inversa de A." << endl;
    return true;
}
```

Pregunta 3

Dado que MIN-COVER es un problema NP-completo, y se requiere una solución en tiempo polinomial con una garantía de que la solución obtenida sea a lo sumo el doble de la óptima (**1-relativo-MIN-COVER** o **2-aproximación**), utilizaremos un algoritmo heurístico que selecciona aristas de forma iterativa.

La estrategia se basa en el principio de que la compuestidad de la solución obtenida (el conjunto de vértices cubiertos) está ligada a la cardinalidad de un apareamiento en el grafo, lo que nos permite establecer la cota de aproximación.

El algoritmo heurístico para el MIN-COVER sin pesos se basa en encontrar un **apareamiento máximo** M (o un buen apareamiento) y usar sus vértices para construir el cubrimiento.

- 1) Inicializar C, el conjunto de cubrimiento de vértices, como vacío.
- 2) Mientras existan aristas sin cubrir en el grafo E:
  - 2.a) Seleccionar una arista no cubierta arbitraria $e = \{u, v\}$.
  - 2.b) Añadir ambos extremos de la arista u y v al conjunto de cubrimiento C.
  - 2.c) Eliminar de la consideración todas las aristas incidentes a u y v (marcándolas como cubiertas).
- 3) Retornar C.

Este algoritmo es de tiempo polinomial. Si se implementa eficientemente (por ejemplo, iterando sobre la lista de aristas originales y marcando los vértices como cubiertos), puede alcanzar una complejidad $O(|V| + |E|)$.

Ahora bien, demostrando la aproximaciónñ se tiene que, sea $G = (N, C)$ el grafo de entrada y sea C^* el conjunto de vértices que constituye un cubrimiento de vértices mínimo ($|C^*| = MIN - COVER$).

- Sea A el conjunto de aristas seleccionadas en el paso 2a del algoritmo antes de que se eliminaran. Por la construcción del algoritmo, una vez que una arista $e = \{u, v\}$ es seleccionada, sus extremos u y v se añaden a C, y todas las aristas incidentes a u o v se marcan como cubiertas. Esto significa que la siguiente arista seleccionada *no* puede ser adyacente a ninguna arista ya seleccionada. Por lo tanto, el conjunto de aristas A es un **apareamiento**.
- Para que C^* sea un cubrimiento, debe cubrir a todas las aristas, incluido el conjunto A. Dado que ninguna arista en A comparte vértices (por definición de apareamiento), C^* debe contener al menos un vértice de cada arista en A.

  $$\text{Por lo tanto: } |C^*| \geq |A|$$

- El algoritmo construye el conjunto C tomando *ambos* extremos de cada arista en A.

  $$\text{Por lo tanto: } |C| = 2 \cdot |A|$$

  Por lo que, combinando las dos relaciones:

  $$|C| = 2 \cdot |A| \leq 2 \cdot |C^*|$$

El algoritmo produce una solución C cuya cardinalidad es a lo sumo el doble de la solución óptima C^*. Por lo tanto, es un **algoritmo de 2-aproximación**.

Implementación en C++

El archivo funcional se encuentra en min_cover.cpp

```cpp
// Definición de tipos para la matriz de adyacencia y el conjunto de vértices
typedef vector<vector<int>> AdjacencyList;
typedef pair<int, int> Edge;

/**
 * @brief Implementa el algoritmo de aproximación 2 para el Cubrimiento Mínimo de Vértices.
 *
 * Este algoritmo encuentra un Cubrimiento de Vértices (C) tal que |C| <= 2 * |C*|,
 * donde C* es el cubrimiento mínimo.
 *
 * @param V Número total de vértices (asumidos indexados de 0 a V-1).
 * @param edge_list Lista de todas las aristas del grafo.
 * @return vector<int> Conjunto de vértices que forman el cubrimiento aproximado.
 */
vector<int> approx_vertex_cover(int V, const vector<Edge>& edge_list) {

    // El conjunto C almacenará los vértices seleccionados para el cubrimiento.
    vector<int> C;

    // 'covered_edge' rastrea si una arista ya ha sido cubierta por los vértices en C.
    // Usamos un set para un acceso rápido y eliminación lógica de aristas cubiertas.
    set<Edge> uncovered_edges(edge_list.begin(), edge_list.end());

    // 'is_in_C' marca qué vértices ya están en el conjunto de cubrimiento C.
    vector<bool> is_in_C(V, false);

    // Iterar mientras queden aristas sin cubrir.
    // Aunque el set 'uncovered_edges' cambia de tamaño, esta sigue siendo una aproximación po
    while (!uncovered_edges.empty()) {
```

```
30
31          // 1. Seleccionar una arista no cubierta arbitraria {u, v}
32          Edge current_edge = *uncovered_edges.begin();
33          int u = current_edge.first;
34          int v = current_edge.second;
35
36          // 2. Añadir ambos extremos al conjunto de cubrimiento C.
37
38          // Solo agregar si el vértice no está ya en C para evitar duplicados en la salida.
39          if (!is_in_C[u]) {
40              C.push_back(u);
41              is_in_C[u] = true;
42          }
43          if (!is_in_C[v]) {
44              C.push_back(v);
45              is_in_C[v] = true;
46          }
47
48          // 3. Marcar/eliminar las aristas incidentes a u y v como cubiertas.
49
50          // Nota: La forma más eficiente de eliminar aristas incidentes
51          // de un set requiere iteradores. Aquí, por simplicidad de implementación,
52          // recreamos el set con solo las aristas que no están cubiertas por C.
53
54          // Podríamos usar un iterador para eliminar aristas, pero para el prototipo:
55          set<Edge> next_uncovered_edges;
56          for (const auto& edge : uncovered_edges) {
57              int x = edge.first;
58              int y = edge.second;
59
60              // Si ninguno de los extremos de la arista está en C, sigue sin cubrir.
61              if (!(is_in_C[x] || is_in_C[y])) {
62                  next_uncovered_edges.insert(edge);
63              }
64          }
65          uncovered_edges = next_uncovered_edges;
66      }
67
68      return C;
69  }
```