**Taller 5**

Estimados,

Para este taller deben comparar la ejecución de procesos paralelos en una problematica basada en matrices:

**Matriz de costos de viajes utilizando backtracking**

Para definir una matriz de costos de distancias de viajes con valores aleatorios en C++, podemos utilizar la librería <random> para generar números aleatorios y la clase vector para almacenar los valores en una matriz.

A continuación, se muestra un ejemplo de cómo crear una matriz de tamaño n con valores aleatorios entre 1 y 10:

|  |
| --- |
| #include <iostream> #include <vector> #include <random>  using namespace std;  int main() {     int n = 5;     vector<vector<int>> matrix(n, vector<int>(n));      random\_device rd;     mt19937 gen(rd());     uniform\_int\_distribution<> dis(1, 10);      for(int i = 0; i < n; i++) {         for(int j = 0; j < n; j++) {             matrix[i][j] = dis(gen);             cout << matrix[i][j] << " ";         }         cout << endl;     }      return 0; } |

Una vez creada la matriz, podemos utilizar backtracking para encontrar el camino más corto desde un punto inicial start hasta un punto final end. Para ello, podemos crear una función backtracking que reciba como parámetros la matriz de costos, el punto actual current, el punto final end, la distancia recorrida hasta el momento dist y una variable para almacenar la distancia mínima minDist. La función deberá explorar todos los caminos posibles desde el punto actual hasta el punto final y actualizar la distancia mínima si se encuentra un camino más corto.

A continuación, se muestra un ejemplo de cómo implementar esta función:

|  |
| --- |
| #include <iostream> #include <vector> #include <random>  using namespace std;  void backtracking(vector<vector<int>>& matrix, int current, int end, int dist, int& minDist, vector<bool>& visited) {     if(current == end) {         minDist = min(minDist, dist);         return;     }      for(int i = 0; i < matrix.size(); i++) {         if(matrix[current][i] != 0 && !visited[i]) {             visited[i] = true;             backtracking(matrix, i, end, dist + matrix[current][i], minDist, visited);             visited[i] = false;         }     } }  int main() {     int n = 5;     vector<vector<int>> matrix(n, vector<int>(n));      random\_device rd;     mt19937 gen(rd());     uniform\_int\_distribution<> dis(1, 10);      for(int i = 0; i < n; i++) {         for(int j = 0; j < n; j++) {             matrix[i][j] = dis(gen);             cout << matrix[i][j] << " ";         }         cout << endl;     }      int start = 0;     int end = 4;     int dist = 0;     int minDist = INT\_MAX;     vector<bool> visited(n, false);     visited[start] = true;      backtracking(matrix, start, end, dist, minDist, visited);      cout << "La distancia mínima desde " << start << " hasta " << end << " es: " << minDist << endl;      return 0; } |

En este ejemplo, se ha creado una matriz de tamaño n=5 con valores aleatorios entre 1 y 10. Se ha definido el punto inicial start=0 y el punto final end=4. Se ha utilizado una variable `

**Solicitud**

Probar y determinar cuando la ejecución paralela se hace efectivamente más eficiente y estable de acuerdo con la condición de cambio de tamaño de la matriz

Sugerencia: Para encontrar el óptimo, comience probando varios casos con matrices de 2X2, luego progresivamente aumente el tamaño de la matriz midiendo los resultados de tiempos de respuesta de procesamiento, apoyándose en un gráfico para obtener el punto de inflexión en donde se hace más eficiente y estable la mejora de la paralelización.

**Compilación**

|  |
| --- |
| **g++ -std=c++0x -o Taller5 Taller5.cpp** |

Nota: En algunos casos el -std puede variar, en donde se puede cambiar a c++11