Instituto Tecnológico de Estudios Superiores Monterrey

Campus Querétaro

[TC2038 – Análisis y Diseño de Algoritmos]

Actividad Integradora 2

Profesora:

Ramona Fuentes Valdez

Presenta:

Ian Joab Padrón Corona	A01708940
Diego Vega Camacho	A01704492
Arturo Cristián Diáz Lopéz	A01709522

Introducción

En el trasfondo dinámico y evolutivo de las ciudades modernas, la eficiencia y la conectividad son pilares fundamentales que definen su progreso. La infraestructura urbana, abarcando desde la red de carreteras hasta las instalaciones logísticas y de comunicación, desempeña un papel crucial en la prosperidad y el funcionamiento armonioso de una comunidad. Este informe se sumerge en el análisis y la resolución de problemáticas clave relacionadas con la planificación y mejora de la infraestructura urbana.

El enfoque adoptado abarca diversas fases, cada una diseñada para abordar un aspecto específico de la optimización urbana. Desde la definición estructural de una ciudad hasta la determinación de rutas logísticas eficientes y la maximización del flujo de información, cada fase contribuye a la creación de un marco integral para la toma de decisiones urbanas informadas.

En primer lugar, se establece la estructura esencial de una ciudad, capturando datos fundamentales sobre sus colonias, conexiones y coordenadas geográficas. A continuación, se aborda la optimización de la conectividad entre colonias mediante el uso del algoritmo de Dijkstra, proporcionando una base sólida para la planificación de la infraestructura de transporte.

La entrega de correspondencia, un elemento vital en la vida urbana, se optimiza a través de la fase de Solución al Problema del Viajante de Comercio (SMP), asegurando rutas logísticas eficientes y la minimización de costos asociados. Además, se explora el Algoritmo Ford-Fulkerson, centrado en maximizar el flujo de información entre las colonias, promoviendo una comunicación eficaz y oportuna.

Finalmente, la expansión y ubicación estratégica de nuevas colonias se abordan mediante el cálculo de la distancia euclidiana, identificando la central más cercana y facilitando decisiones informadas sobre la expansión de la infraestructura central.

Fase 1: Estructura principal de una ciudad

En esta fase, se establece la base estructural del programa mediante la definición de la estructura City. Esta estructura encapsula información crucial sobre una ciudad, incluyendo el número de colonias, las distancias entre ellas, el flujo máximo de información, y las coordenadas geográficas de cada colonia. Al organizar estos datos de manera cohesiva, se sienta el fundamento para abordar problemáticas más complejas de la infraestructura.

```cpp
// ============================================================================
// Estructura City, almacena los datos de una ciudad
//
// @params nCities: Número de colonias en la ciudad
// @params distBtwnClnies: Distancias en kms entre las colonias de la ciudad
// @params maxFluxBtwnClnies: Flujo máximo entre las colonias de la ciudad
// @params coordsClnies: Coordenadas de las colonias de la ciudad
// @params coordsNewClny: Coordenadas de la nueva colonia
// ============================================================================

struct City {

    int nCities;
    vector<vector<int>> distBtwnClnies;
    vector<vector<int>> maxFluxBtwnClnies;
    vector<pair<int, int>> coordsClnies;
    pair<int, int> coordsNewClny;

    City(int n) :
        nCities(n),
        distBtwnClnies(n, std::vector<int>(n, 0)),
        maxFluxBtwnClnies(n, std::vector<int>(n, 0)),
        coordsClnies(n, std::pair<int, int>(0, 0)),
        coordsNewClny(0, 0) {}

    friend std::ostream &operator<<(std::ostream &os, const City &city) {
        os << "distBtwnClnies:" << endl;
        for (const auto &row : city.distBtwnClnies) {
            for (const auto &elem : row) {
                os << elem << ' ';
            }
            os << std::endl;
        }

        os << "maxFluxBtwnClnies:" << endl;
        for (const auto &row : city.maxFluxBtwnClnies) {
            for (const auto &elem : row) {
                os << elem << ' ';
            }
            os << endl;
        }

        os << "coordsClnies:" << endl;
        for (const auto &coord : city.coordsClnies) {
            os << '(' << coord.first << ", " << coord.second << ')' << endl;
        }

        os << "coordsNewClny: (" << city.coordsNewClny.first << ", " << city.coordsNewClny.second << ')' << endl;

        return os;
    }
};
```

Fase 2: Lectura de archivos

En esta fase, se implementa la función readFromFile, la cual permite la lectura de información clave desde un archivo de texto. Esta función facilita la entrada de datos, permitiendo que el programa trabaje con información realista y específica de cada ciudad, adaptándose así a diversas situaciones y escenarios.

```cpp
// Función readFromFile, lee los datos de un archivo de texto y los almacena en una estructura
// de datos
//
// @params filename: Nombre del archivo de texto
//
// @return: Regresa una estructura de datos con los datos de la ciudad
// @complexity O(n)
// ================================================================================

City readFromFile(const string &filename) {

    ifstream inputFile(filename);
    if (!inputFile.is_open()) {
        cout << "No se pudo abrir el archivo" << endl;
        exit(EXIT_FAILURE);
    }

    int n;
    inputFile >> n;

    City city(n);
    city.nCities = n;

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            inputFile >> city.distBtwnClnies[i][j];
        }
    }

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            inputFile >> city.maxFluxBtwnClnies[i][j];
        }
    }

    for (int i = 0; i < n + 1; ++i) {
        string linea = "";
        inputFile >> linea;

        stringstream ss(linea);

        char paren, comma;
        int x, y;
        ss >> paren >> x >> comma >> y >> paren;

        (i < n) ? city.coordsClnies[i] = pair<int, int>(x, y) : city.coordsNewClny = pair<int, int>(x, y);

    }

    return city;
}
```

Fase 3: Encontrar la forma óptima de cablear las colonias (Dijkstra)

Para obtener la conectividad entre las colonias de una ciudad, se aplica el algoritmo de Dijkstra. Este algoritmo, implementado en la función dijkstra, encuentra de manera eficiente las distancias mínimas entre una colonia y todas las demás. La resultante red de distancias mínimas sirve como base para la planificación de la infraestructura, contribuyendo a una conectividad eficiente y económica.

```cpp
// ============================================================================================
// Función dijkstra, implementación del algoritmo de Dijkstra para encontrar la distancia
// mínima entre una colonia y todas las demás
//
// @params n: Número de colonias en la ciudad
// @params graph: Distancias en kms entre las colonias de la ciudad
// @params city: Colonia a la que se le aplicará el algoritmo
//
// @complexity O(n)
// ============================================================================================

void dijkstra(int n, const vector<vector<int>> &graph, int city) {
    vector<int> dist(n, INT_MAX);
    vector<bool> sptSet(n, false);

    dist[city] = 0;

    for (int count = 0; count < n - 1; count++) {
        int u = minDistance(n, dist, sptSet);

        sptSet[u] = true;

        for (int v = 0; v < n; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX &&
                dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    printSolutionDijkstra(city, n, dist);
}
```

```cpp
// ================================================================================================
// Función minDistance, encuentra la distancia mínima entre una colonia y todas las demás
//
// @params n: Número de colonias en la ciudad
// @params dist: Distancias en kms entre las colonias de la ciudad
// @params sptSet: Arreglo de booleanos que indica si una colonia ya fue visitada
//
// @return: Regresa el índice de la colonia más cercana
// @complexity O(n)
// ================================================================================================

int minDistance(int n, const vector<int> &dist, const vector<bool> &sptSet) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < n; v++) {
        if (!sptSet[v] && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }

    return min_index;
}
```

```cpp
// ================================================================================================
// Función printSolutionDijkstra, imprime la distancia mínima entre una colonia y todas las
// demás
//
// @params city: Colonia a la que se le aplicó el algoritmo
// @params n: Número de colonias en la ciudad
// @params dist: Distancias en kms entre las colonias de la ciudad
//
// @complexity O(n)
// ================================================================================================

void printSolutionDijkstra(int city, int n, const vector<int> &dist) {
    for (int i = 0; i < n; i++) {
        if (i != city) {
            cout << "Colonia " << city + 1 << " a colonia " << i + 1 << ": " <<  dist[i] << endl;
        }
    }
}
```

Fase 4: Encontrar la ruta más corta para que el personal entregue correspondencia (Salesman Problem)

La eficiencia en la entrega de correspondencia es un aspecto clave de la logística urbana. La fase de Solución al Problema del Viajante de Comercio (SMP), implementada en la función smp, permite determinar la ruta más corta que permite al personal recorrer todas las colonias. Este algoritmo no solo optimiza la logística, sino que también contribuye a la reducción de costos asociados con el desplazamiento.

```
// ================================================================================
// Función findNearestCity, encuentra la colonia más cercana a una colonia dada
//
// @params numCities: Número de colonias en la ciudad
// @params distanceMatrix: Distancias en kms entre las colonias de la ciudad
// @params city: Colonia a la que se le aplicará el algoritmo
// @params visited: Arreglo de booleanos que indica si una colonia ya fue visitada
//
// @return: Regresa el índice de la colonia más cercana
// @complexity O(n)
// ================================================================================

int findNearestCity(int numCities, const vector<vector<int>> &distanceMatrix, int city, vector<bool> &visited) {
    int minDistance = INT_MAX;
    int nearestCity = -1;

    for (int i = 0; i < numCities; ++i) {
        if (!visited[i] && distanceMatrix[city][i] < minDistance) {
            minDistance = distanceMatrix[city][i];
            nearestCity = i;
        }
    }

    return nearestCity;
}
```

```cpp
// ============================================================================================
// Función smp, implementación del algoritmo de la colonia mas cercana para encontrar el
// recorrido mas corto que pase por todas las colonias de una ciudad
//
// @params numCities: Número de colonias en la ciudad
// @params distanceMatrix: Distancias en kms entre las colonias de la ciudad
//
// @return: Regresa el costo total del recorrido
// @complexity O(n)
// ============================================================================================

int smp(int numCities, const vector<vector<int>> &distanceMatrix) {
    vector<bool> visited(numCities, false);
    vector<int> path;
    int totalDistance = 0;

    int currentCity = 0;
    path.push_back(currentCity + 1);
    visited[currentCity] = true;

    for (int i = 0; i < numCities; ++i) {
        int nearestCity = findNearestCity(numCities, distanceMatrix, currentCity, visited);
        if (nearestCity != -1) {
            path.push_back(nearestCity + 1);
            visited[nearestCity] = true;
            totalDistance += distanceMatrix[currentCity][nearestCity];
            currentCity = nearestCity;
        }
    }
    path.push_back(path[0]);
    totalDistance += distanceMatrix[currentCity][path[0] - 1];

    cout << "El recorrido:" << endl;
    for (size_t i = 0; i < path.size() - 1; ++i) {
        cout << path[i] << " -> ";
    }
    cout << path.back() << endl;

    cout << "El costo: " << totalDistance << endl;

    return totalDistance;
}
```

Fase 5: Calcular el flujo máximo de información (Ford-Fulkerson)

El intercambio de información entre colonias es esencial para el funcionamiento eficiente de una ciudad. La fase del Algoritmo de Ford-Fulkerson, implementada en la función fordFulkerson, aborda la maximización del flujo de información entre dos nodos de un grafo. Al modelar la red de colonias y conexiones como un flujo de red, esta fase contribuye a optimizar la capacidad de comunicación dentro de la ciudad.

```cpp
// ============================================================================================
// Función bfs, encuentra el camino más corto entre dos nodos de un grafo
//
// @params n: Número de colonias en la ciudad
// @params rfluxMatrix: Flujo residual entre las colonias de la ciudad
// @params s: Colonia de origen
// @params t: Colonia de destino
// @params parent: Arreglo que almacena el camino más corto
//
// @return: Regresa true si existe un camino entre el nodo de origen y el nodo de destino
// @complexity O(n)
// ============================================================================================

bool bfs(int n, const vector<vector<int>> &rfluxMatrix, int s, int t, int parent[]) {

    vector<bool> visited(n, false);

    queue<int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v = 0; v < n; v++) {
            if (visited[v] == false && rfluxMatrix[u][v] > 0) {
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }

    return (visited[t] == true);
}
```

```cpp
// ============================================================================================
// Función fordFulkerson, implementación del algoritmo de Ford-Fulkerson para encontrar el
// flujo máximo entre dos nodos de un grafo
//
// @params n: Número de colonias en la ciudad
// @params fluxMatrix: Flujo entre las colonias de la ciudad
// @params s: Colonia de origen
// @params t: Colonia de destino
//
// @return: Regresa el flujo máximo entre el nodo de origen y el nodo de destino
// @complexity O(n)
// ============================================================================================

int fordFulkerson(int n, const vector<vector<int>> &fluxMatrix, int s, int t) {
    int u, v;

    vector<vector<int>> rfluxMatrix(n, vector<int>(n, 0));
    for (u = 0; u < n; u++)
        for (v = 0; v < n; v++)
            rfluxMatrix[u][v] = fluxMatrix[u][v];

    int parent[n];
    int max_flow = 0;

    while (bfs(n, rfluxMatrix, s, t, parent)) {
        int path_flow = INT_MAX;
        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            path_flow = min(path_flow, rfluxMatrix[u][v]);
        }

        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            rfluxMatrix[u][v] -= path_flow;
            rfluxMatrix[v][u] += path_flow;
        }

        max_flow += path_flow;
    }

    return max_flow;
}
```

Fase 6: Encontrar la central más cercana

La expansión urbana a menudo implica la adición de nuevas colonias y la ubicación estratégica de centrales. En esta fase, se emplea la distancia euclidiana para identificar la central más cercana a una nueva colonia. La función findNearestCentral evalúa las coordenadas geográficas de las centrales existentes y determina cuál está en la proximidad más cercana, facilitando decisiones informadas sobre la expansión de la infraestructura central.

```cpp
// ============================================================================================
// Función distanceBetweenColonies, calcula la distancia euclidiana entre dos colonias
//
// @params colony1: Par ordenado que representa la ubicación en un plano de la primera colonia
// @params colony2: Par ordenado que representa la ubicación en un plano de la segunda colonia
//
// @return: Regresa la distancia euclidiana entre las dos colonias
// @complexity O(1)
// ============================================================================================

double distanceBetweenColonies(const pair<int, int> &colony1, pair<int, int> &colony2) {
    return sqrt(pow(colony1.first - colony2.first, 2) + pow(colony1.second - colony2.second, 2));
}
```

```cpp
// ============================================================================================
// Función findNearestCentral, encuentra la central mas cercana a una colonia
//
// @params newCentral: Par ordenado que representa la ubicación en un plano de la nueva
// central
// @params centralLocations: Pares ordenados que representan la ubicación en un plano de las
// centrales
//
// @return: Índice de la central mas cercana a la colonia utilizando la distancia euclidiana
//
// @complexity
// ============================================================================================

int findNearestCentral(pair<int, int> &newCentral, vector<pair<int, int>> &centralLocations) {
    int nearestCentral = -1;
    pair<int, int> nearestCentralCoords;
    double minDistance = INT_MAX;

    for (int i = 0; i < centralLocations.size(); i++) {
        double distance = distanceBetweenColonies(newCentral, centralLocations[i]);
        if (distance < minDistance)
        {
            minDistance = distance;
            nearestCentral = i;
            nearestCentralCoords = centralLocations[i];
        }
    }

    printf("La central mas cercana a [%d,%d] es [%d,%d] con una distancia de %.3f.\n", newCentral.first,
    newCentral.second, nearestCentralCoords.first, nearestCentralCoords.second, minDistance);
    cout << endl;

    return nearestCentral;
}
```

Conclusiones

En conjunto, este reporte subraya la necesidad de adoptar un enfoque integral en la planificación urbana. La colaboración entre disciplinas, la aplicación de algoritmos avanzados y el uso de datos precisos forman la base para construir ciudades más adaptativas a nuestras necesidades cambiantes con el paso del tiempo. A medida que las ciudades evolucionan, la optimización de la infraestructura se convierte en un imperativo, allanando el camino hacia entornos urbanos más sostenibles, resilientes y preparados para las demandas del futuro.