INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

AI2_Actividad Integradora 2 (subir programa en equipo)

Curso: TC2038 **Grupo:** 601

Integrantes:

Daniel Emilio Fuentes Portaluppi - A01708302
Daniel Sebastián Cajas Morales - A01708637
Diego Ernesto Sandoval Vargas - A01709113

Profesor: Ramona Fuentes Valdéz

Link video:
https://drive.google.com/file/d/1ZsExaVmA80uAjDjiOg5lT1ZRZGueoLoG/view?usp=sharing

Noviembre 18, 2023

Situación problema 2

Alta demanda para los Proveedores de Servicios de Internet (ISP)

Durante el año 2020 todo el mundo se vio afectado por un evento que nadie esperaba: la pandemia ocasionada por el COVID-19. En todos los países del planeta se tomaron medidas sanitarias para intentar contener la pandemia. Una de estas medidas fue el mandar a toda la población a sus casas, moviendo gran parte de las actividades presenciales a un modelo remoto en el que las empresas proveedoras de servicios de Internet (ISP por sus siglas en inglés de Internet Service Provider) tomaron un papel más que protagónico. Mucha gente se movió a la modalidad de trabajo remoto, o home-office, también la mayoría de instituciones educativas optaron por continuar sus operaciones bajo un modelo a distancia aumentando de gran forma la transmisión de datos en Internet.

Si estuviera en nuestras manos mejorar los servicios de Internet en una población pequeña,

¿Podríamos decidir cómo cablear los puntos más importantes de dicha población de tal forma que se utilice la menor cantidad de fibra óptica?

Asumiendo que tenemos varias formas de conectar dos nodos en la población,

Para una persona que tiene que ir a visitar todos los puntos de la red, ¿Cuál será la forma óptima de visitar todos los puntos de la red y regresar al punto de origen?

¿Podríamos analizar la cantidad máxima de información que puede pasar desde un nodo a otro?

¿Podríamos analizar la factibilidad de conectar a la red un nuevo punto (una nueva localidad) en el mapa ?

Parte 1

Leer un archivo de entrada que contiene la información de un grafo representado en forma de una matriz de adyacencias con grafos ponderados.
El peso de cada arista es la distancia en kilómetros entre colonia y colonia, por donde es factible meter cableado.
El programa debe desplegar cuál es la forma óptima de cablear con fibra óptica conectando colonias de tal forma que se pueda compartir información entre cualesquiera dos colonias.

Solución

Para esta parte se utilizó el algoritmo de Kruskal el cual busca unir todos los nodos tomando en cuenta el peso de las aristas y buscando el coste total mínimo, para esto el algoritmo selecciona aristas de menor peso, evitando ciclos mediante la gestión de conjuntos disjuntos

Complejidad

La complejidad del algoritmo de kruskas es de *O(NlogN)* donde N representa el número de aristas, por lo que es un algoritmo el cual sigue siendo bastante eficiente inclusive si aumentamos la cantidad de nodos a ordenar. la complejidad puede variar según la implementación específica de las estructuras de datos utilizadas para conjuntos disjuntos y la cola de prioridad.

Capturas del código

```cpp
struct Edge
{
    int weight;
    int u;
    int v;
};

bool operator>(const Edge &lhs, const Edge &rhs)
{
    return lhs.weight > rhs.weight;
}

template <class T>
using i_priority_queue = priority_queue<T, vector<T>, greater<T>>;

int find(int i, vector<int> &parent)
{
    if (parent[i] == -1)
        return i;

    return parent[i] = find(parent[i], parent);
}

void Union(int rootI, int rootJ, vector<int> &parent, vector<int> &rank)
{
    if (rootI != rootJ)
    {
        if (rank[rootI] < rank[rootJ])
            parent[rootI] = rootJ;
        else if (rank[rootI] > rank[rootJ])
            parent[rootJ] = rootI;
        else
        {
            parent[rootJ] = rootI;
            rank[rootI]++;
        }
    }
}
```

```
WGraph mst(const WGraph &graph)
{
    int numVertices = graph.getNumVertices();
    vector<Edge> result;
    WGraph mst(numVertices);

    // Initialize parent and rank arrays
    vector<int> parent(numVertices, -1);
    vector<int> rank(numVertices, 1);

    // Priority queue to store edges sorted by weight
    i_priority_queue<Edge> pq;

    // Add all edges to the priority queue
    for (int i = 0; i < numVertices; ++i)
    {
        for (int j : graph.getNeighbours(i))
        {
            int weight = graph.getWeight(i, j);
            pq.push({weight, i, j});
        }
    }

    // Process edges from the priority queue
    while (!pq.empty())
    {
        Edge edge = pq.top();
        pq.pop();

        // Check for a cycle
        int rootI = find(edge.u, parent);
        int rootJ = find(edge.v, parent);
        if (rootI != rootJ)
        {
            mst.addEdge(edge.u, edge.v, edge.weight);
            Union(rootI, rootJ, parent, rank);
        }
    }
    return mst;
}
```

Parte 2

Debido a que las ciudades apenas están entrando al mundo tecnológico, se requiere que alguien visite cada colonia para ir a dejar estados de cuenta físicos, publicidad, avisos y notificaciones impresos. por eso se quiere saber ¿cuál es la ruta más corta posible que visita cada colonia exactamente una vez y al finalizar regresa a la colonia origen?

El programa debe desplegar la ruta a considerar, tomando en cuenta que la primera ciudad se le llamará A, a la segunda B, y así sucesivamente.

Solución

Para esta situación se utilizó el algoritmo de fuerza bruta que resuelve el problema de TSP, este sería una de las soluciones a este problema. Este algoritmo considera todas los posibles caminos de los nodos y calcula el costo de cada camino hasta encontrar el más óptimo.

Complejidad

Este algoritmo tiene una complejidad de $O(n!)$ por lo que en situaciones con muchos datos, aunque es una solución a este problema, puede llegar a no ser la más eficiente.

Capturas del código

```
pair<int, vector<char>> tsp(WGraph graph, int s)
{
    vector<int> vertex;
    for (int i = 0; i < graph.getNumVertices(); i++)
        if (i != s)
            vertex.push_back(i);

    int minCost = INT_MAX;
    vector<char> minPath;

    do
    {
        int currentCost = 0;
        vector<char> path = {static_cast<char>('A' + s)};
        int k = s;

        for (int i = 0; i < vertex.size(); i++)
        {
            currentCost += graph.getWeight(k, vertex[i]);
            k = vertex[i];
            path.push_back(static_cast<char>('A' + vertex[i]));
        }
        currentCost += graph.getWeight(k, s);

        if (currentCost < minCost)
        {
            minCost = currentCost;
            minPath = path;
        }

    } while (next_permutation(vertex.begin(), vertex.end()));

    return {minCost, minPath};
}
```

Parte 3

El programa también debe leer otra matriz cuadrada de N x N datos que representen la capacidad máxima de transmisión de datos entre la colonia i y la colonia j. Como estamos trabajando con ciudades con una gran cantidad de campos electromagnéticos, que pueden generar interferencia, ya se hicieron estimaciones que están reflejadas en esta matriz.

La empresa quiere conocer el flujo máximo de información del nodo inicial al nodo final. Esto debe desplegarse también en la salida estándar.

Solución

Para esta situación se utilizó el algoritmo de Ford-Fulkerson, que propone buscar caminos en los que se pueda aumentar el flujo, hasta que se alcance el flujo máximo. Para esto se hace una búsqueda de todos los posibles caminos donde aún exista flujo disponible y se actualiza un grafo residual, hasta que no existan más caminos posibles. Aquí sumamos todos los flujos que entran a nuestro nodo de salida y ese es nuestro flujo final.

Complejidad

La complejidad de este algoritmo es de $O(max\_flow * E)$. Ya que en el peor de los casos, ese aumento en una unidad cada iteración al max flow y cada iteración tuvo que revisar E edges o aristas. Esto implica que mientras más conexiones tengamos más tiempo tomará, pero también los pesos de estas conexiones y su distribución contribuye, haciéndolo difícil de predecir.

Capturas del código

```
bool bfs(WGraph rGraph, int start, int end, int parent[])
{
    int nVertexes = rGraph.getNumVertices();
    bool visited[nVertexes];
    for (int i = 0; i < rGraph.getNumVertices(); ++i)
        visited[i] = false;

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

    while (!q.empty())
    {
        int current = q.front();
        q.pop();
        auto neighbours = rGraph.getNeighboursWithWeight(current);
        for (auto neighbour : neighbours)
        {
            if (visited[neighbour.first] == false && neighbour.second > 0)
            {
                if (neighbour.first == end)
                {
                    parent[neighbour.first] = current;
                    return true;
                }
                q.push(neighbour.first);
                parent[neighbour.first] = current;
                visited[neighbour.first] = true;
            }
        }
    }
    return false;
}

```

```
pair<int, WGraph> fordFulkerson(WGraph graph, int start, int end)
{
    int nVertexes = graph.getNumVertices();
    WGraph rGraph(graph);
    int parent[nVertexes];
    int max_flow = 0;

    while (bfs(rGraph, start, end, parent))
    {
        int path_flow = INT_MAX;
        int current = end;
        while (current != start)
        {
            int u = parent[current];
            path_flow = min(path_flow, rGraph.getWeight(u, current));
            current = parent[current];
        }

        current = end;
        while (current != start)
        {
            int u = parent[current];
            rGraph.setWeight(u, current, rGraph.getWeight(u, current) - path_flow);
            rGraph.setWeight(current, u, rGraph.getWeight(current, u) + path_flow);
            current = parent[current];
        }

        max_flow += path_flow;
    }
    return {max_flow, rGraph};
}
```

Parte 4

Teniendo en cuenta la ubicación geográfica de varias "centrales" a las que se pueden conectar nuevas casas, la empresa quiere contar con una forma de decidir, dada una nueva contratación del servicio, cuál es la central más cercana geográficamente a esa nueva contratación. No necesariamente hay una central por cada colonia. Se pueden tener colonias sin central, y colonias con más de una central.

Investigación

Nearest Neighbor

El algoritmo de Nearest Neighbor (Vecino más Cercano), también conocido como K-NN (K-Nearest Neighbors), es un algoritmo de aprendizaje automático que se basa en la premisa de que puntos cercanos tendrán características o valores de salida similares. Este algoritmo se utiliza comúnmente en problemas de clasificación y regresión, pero también puede aplicarse a situaciones en las que se busca encontrar la ubicación más cercana a un punto dado.

En el contexto de la ubicación geográfica de centrales y nuevas contrataciones de servicios, el algoritmo Nearest Neighbor puede ser una herramienta eficaz para determinar qué central está más cerca de una ubicación específica.

Complejidad

Existen diferentes métodos para implementar el algoritmo por lo que su complejidad varía según el método utilizado:

Fuerza Bruta

Complejidad del entrenamiento: $O(1)$

Complejidad de Predicción: $O(k * n * d)$

donde:

- K: Cantidad de Vecinos
- n: Número de Puntos
- d: Dimensiones de la información

En este método la complejidad de entrenamiento permanece en $O(1)$ puesto que no hay entrenamiento y todo el proceso de predicción se hace mientras se predice.

k-d tree

Complejidad del entrenamiento: *O(d * n * log(n))*

Complejidad de Predicción: *O(k * log(n))*

Donde:

- K: Cantidad de Vecinos
- n: Número de Puntos
- d: Dimensiones de la información

En este método se utiliza una estructura de árbol KD (k dimensiones) para entrenar el modelo, lo que aunque genera complejidad al entrenar hacer que el modelo resultante para predecir tenga una menor complejidad.

Solución

Para la implementación como prueba de concepto decidimos implementar el algoritmo de fuerza bruta.

Complejidad

*O(k * n * d)*

Capturas del código

```
pair<int, int> nearest_neighbour(vector<pair<int, int>> &points, pair<int, int> &targuet){
    // brute force
    float min_dist = INT_MAX;
    pair<int, int> min_pair;

    for (auto point : points){
        float dist = sqrt(pow(point.first - targuet.first, 2) + pow(point.second - targuet.second, 2));
        if (dist < min_dist){
            min_dist = dist;
            min_pair = point;
        }
    }

    return min_pair;
}
```

Función Main

```
int main()
{
    cout << "Number of vertices: ";
    int nVertexes;
    cin >> nVertexes;

    WGraph graph(nVertexes);

    cout << "\nAdjacency Matrix of Distances: " << endl;

    for (int i = 0; i < nVertexes; i++)
    {
        cout << i + 1 << ": ";
        for (int j = 0; j < nVertexes; j++)
        {
            int weight;
            cin >> weight;
            if (weight != -1)
            {
                graph.addEdge(i, j, weight);
            }
        }
    }
```

```
auto tspAns = tsp(graph, 0);

cout << "\nCosto mínimo: " << tspAns.first << endl;
cout << "Recorrido más eficiente: ";
for (auto i : tspAns.second)
    cout << i << " ";
cout << endl;

WGraph graph2(nVertexes);
cout << "\nAdjacency Matrix of Max throughput: " << endl;

for (int i = 0; i < nVertexes; i++)
{
    cout << i + 1 << ": ";
    for (int j = 0; j < nVertexes; j++)
    {
        int weight;
        cin >> weight;
        if (weight != -1)
        {
            graph2.addEdge(i, j, weight);
        }
    }
}

cout << "Input Graph of Max throughput:\n"
     << graph2.toString() << endl;
```

```
auto ans = fordFulkerson(graph2, 0, nVertexes - 1);
    cout << "Max Flow: " << ans.first << endl;

    cout << "Residual Graph: " << endl;
    cout << ans.second.toString() << endl;

    cout << "Nearset Neighbour: " << endl;
    vector<pair<int, int>> points(nVertexes);

    for (int i = 0; i < nVertexes; i++)
    {
        cout << "Point " << i + 1 << ": ";
        string buffer;
        // parse (###, ###) into numbers
        cin >> buffer;
        buffer = buffer.substr(1, buffer.size() - 2);
        int comma = buffer.find(',');
        points[i].first = stoi(buffer.substr(0, comma));
        points[i].second = stoi(buffer.substr(comma + 1, buffer.size() - comma - 1));
    }

    cout << "Targuet: ";
    string buffer;
    // parse (###, ###) into numbers
    cin >> buffer;
    buffer = buffer.substr(1, buffer.size() - 2);
    int comma = buffer.find(',');
    pair<int, int> targuet;
    targuet.first = stoi(buffer.substr(0, comma));
    targuet.second = stoi(buffer.substr(comma + 1, buffer.size() - comma - 1));

    auto ans2 = nearest_neighbour(points, targuet);

    cout << "Nearest Neighbour: " << ans2.first << ", " << ans2.second << endl;

    return 0;
}
```

Output

```
❯ ./app
Number of vertices: 4

Adjacency Matrix of Distances:
1: 0 16 45 32
2: 16  0 18 21
3: 45 18  0  7
4: 32 21  7  0
Input Graph of Distances:
WGraph: 4 vertices, 16 edges
        1       2       3       4
1       0       16      45      32
2       16      0       18      21
3       45      18      0       7
4       32      21      7       0

MST:
WGraph: 4 vertices, 3 edges
        1       2       3       4
1       0       16      0       0
2       0       0       18      0
3       0       0       0       7
4       0       0       0       0

Costo mínimo: 73
Recorrido más eficiente: A B C D

Adjacency Matrix of Max throughput:
1: 0 48  12  18
2: 52  0 42 32
3: 18 46  0 56
4: 24 36 52  0
Input Graph of Max throughput:
WGraph: 4 vertices, 16 edges
        1       2       3       4
1       0       48      12      18
2       52      0       42      32
3       18      46      0       56
4       24      36      52      0
Max Flow: 78
Residual Graph:
WGraph: 4 vertices, 16 edges
        1       2       3       4
1       0       0       0       0
2       100     0       26      0
3       30      62      0       28
4       42      68      80      0
Nearset Neighbour:
Point 1: (200,500)
Point 2: (300,100)
Point 3: (450,150)
Point 4: (520,480)
Targuet: (325,200)
Nearest Neighbour: 300, 100
```

Conclusión

Situaciones cómo las que se presentan en la Situación Problema, son situaciones que se viven día tras día en un mundo tan conectado como el nuestro. La eficiencia de la conexión dependerá de gran manera de una exhaustiva preparación para las conexiones de la red.
Es por eso que existen algoritmos cómo los elaborados que nos ayudan a darle solución a estas problemáticas. Buscando la manera de usar el menos cable posible como en el problema uno, o cual la vía más rápida para conectar las distintas localidades cómo en el punto dos, también obtener la mayor cantidad de datos en el punto tres y cuál es la ruta más corta para conectarse a una central eléctrica.
Aunque esto parecerían problemas de electrónica o planeación, pueden ser resueltos también con código y grafos.

Referencias

KeepCoding, R. (2022, 25 noviembre). ¿Cómo funciona el algoritmo de Vecinos Más Próximos o K-NN? *KeepCoding Bootcamps*. https://keepcoding.io/blog/algoritmo-de-vecinos-mas-proximos/#Usos_del_algoritmo_de_vecinos_mas_proximos

Adamczyk, J. (2022, 30 marzo). K Nearest neighbors Computational complexity - towards data science. *Medium*. https://towardsdatascience.com/k-nearest-neighbors-computational-complexity-502d2c440d5