Instituto Tecnológico de Estudios Superiores de Monterrey

Campus Ciudad de México

TC2038.651 Diseño y Análisis de Algoritmos Avanzados

Actividad Integradora 2

Alumnos

Alejandro Díaz Villagómez | A01276769

Emiliano Saucedo Arriola | A01659258

Profesor

Sergio Ruiz Loza

Fecha de entrega

17 de noviembre de 2022

Introducción

Hoy en día, existen diferentes herramientas matemáticas que nos permiten modelar sistemas en donde existen relaciones entre entidades para encontrar soluciones óptimas a problemas complejos de la vida cotidiana. La teoría de grafos es una rama de la matemática que ofrece la posibilidad de resolver diversos problemas mediante la representación visual de datos abstractos y la relación que hay entre ellos a través de nodos y aristas. Por mencionar algunas de sus aplicaciones, es posible enumerar el uso de grafos para la determinación de la ruta óptima que un transportista debe utilizar en el envío de paquetes o en las redes sociales, donde estos pueden ser de provecho para encontrar patrones o preferencias entre los usuarios. De igual forma, la geometría computacional resulta relevante para brindar soluciones a problemas geométricos haciendo uso eficiente de algoritmos y estructura de datos. Una de sus aplicaciones es la búsqueda geométrica computacional, ya sea para buscar vecinos cercanos a un determinado punto o rango o la localización de puntos en el espacio.

Por lo anterior, en esta evidencia se busca poner en práctica los conocimientos adquiridos sobre teoría de grafos y búsqueda geométrica para analizar una problemática de la vida cotidiana en la que se desea incorporar una región geográfica al mundo tecnológico. Para lograr lo anterior, es necesario determinar diferentes aspectos, como la manera de realizar un cableado de forma eficiente entre las ciudades, identificar la mejor ruta para que los trabajadores puedan realizar sus operaciones dentro del espacio físico, calcular el flujo máximo de información que puede transmitirse y tomar decisiones con respecto a la proximidad de las centrales dado una contratación nueva.

Descripción de la evidencia

En equipos de máximo 3 personas,
Escribe en C++ un programa que ayude a una empresa que quiere incursionar en los servicios de Internet respondiendo a la situación problema 2. El programa debe:

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 cuales quiera dos colonias.

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.

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.

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.

Para comprobar nuestro código, recibiremos un archivo txt con la siguiente información:

Input (CasoPrueba.txt)

- Un numero entero N que representa el número de colonias en la ciudad
- Matriz cuadrada de N x N que representa el grafo con las distancias en kilómetros entre las colonias de la ciudad
- Matriz cuadrada de N x N que representa las capacidades máximas de flujo de datos entre colonia i y colonia j
- Lista de N pares ordenados de la forma (A,B) que representan la ubicación en un plano coordenado de las centrales

Solución

La Situación Problema se divide en 4 etapas:

1) Cableado eficiente entre colonias

Para poder determinar la manera más eficiente de realizar el cableado, es posible tener un acercamiento a través de grafos. Existen diferentes métodos para poder resolverlo.

El primer acercamiento que se tuvo fue a través del algoritmo de Floyd-Warshall, el cual permite encontrar la distancia más corta en un grafo dirigido ponderado. A diferencia de Dijkstra, este método se apoya fuertemente en matrices (sacando provecho a la programación dinámica) y busca el camino mínimo para todos los nodos de un grafo. Por otra parte, Dijkstra encuentra el mejor camino para un nodo en el grafo haciendo un recorrido DFS (Depth First Search).

Es posible generar el mismo resultado de Floyd Warshall con Dijkstra, es decir, calculando el mejor camino para todos los nodos, sin embargo, considerando la complejidad de la implementación realizada en tareas/ejercicios previos ($O(n^2)$), el realizar esta modificación se traduce en una complejidad de $O(n^3)$. Si bien, la complejidad entre ambos algoritmos es la misma, inicialmente, optamos por el método de Floyd-Warshall debido a la naturaleza de su enfoque de obtener el camino menos costoso para todos los nodos.

A pesar de que esta solución es válida, como se describió, la complejidad del algoritmo es muy costosa (es necesario usar un triple ciclo for anidado). Por ello, es conveniente buscar alguna alternativa más eficiente: algoritmo de Kruskal.

El método de Kruskal permite obtener un subgrafo conformado por las aristas de peso mínimo que conforman al grafo original. Debido a esto, Kruskal permite obtener cuáles serían las aristas (o conexiones) entre ciudades que poseen la menor distancia entre sí para llevar a cabo un cableado eficiente. Por cuestiones de abstracción, es necesario crear un archivo adicional para realizar las operaciones de subconjuntos necesarias para este algoritmo. La complejidad resultante es de $O(n^2 * m)$, la cuál es sustancialmente mejor que cualquier solución previamente planteada.

Código

Floyd-Warshall

$O(n^3)$

```cpp
//-------------------------------------------------------------------------------------------------FLOYD-WARSHALL
//FUNCIÓN FLOYD-WARSHALL: Calcula el camino más corto con el algoritmo Floyd-Warshall
//Complejidad --> O(n^3)
void Graph::FloydWarshall(){
    //Preparamos la matriz
    vector<vector<int>> dist;
    int v = (int)nodes.size();

    //Complejidad --> O👎
    for (int i = 0; i < v; i++){
        vector<int> row(v, INFINITO);
        row[i] = 0; //La diagonal queda en 0
        dist.push_back(row);
    }

    //Inicializamos los valores correspondientes a los edges existentes
    //Complejidad --> O(e)
    for (Edge* e : edges){
        dist[e->first->number - 1][e->second->number - 1] = e->weight;
    }

    //Crearemos v matrices para calcular el resultado
    //Además necesitamos 2 ciclos adicionales para iterar la matriz
    //Complejidad --> O(n^3)
    for (int k = 0; k < v; k++){
        for (int i = 0; i < v; i++){
            for (int j = 0; j < v; j++){
                if (dist[i][k] != INFINITO && dist[k][j] != INFINITO &&
                    dist[i][k] + dist[k][j] < dist[i][j] )
                 dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    //Impresión del resultado
    //Complejidad --> O(n^2)
    for (int i = 0; i < v; i++){
        for (int j = 0; j < v; j++){
            if (dist[i][j] == INFINITO or dist[i][j] == 0) continue;
            else cout << "\tArco: (" << to_string(i + 1) << ", " << to_string(j + 1)
                << ")\t\tDistancia: " << dist[i][j] << "\tKm\n";
        }
    }
    cout << endl;
}
```

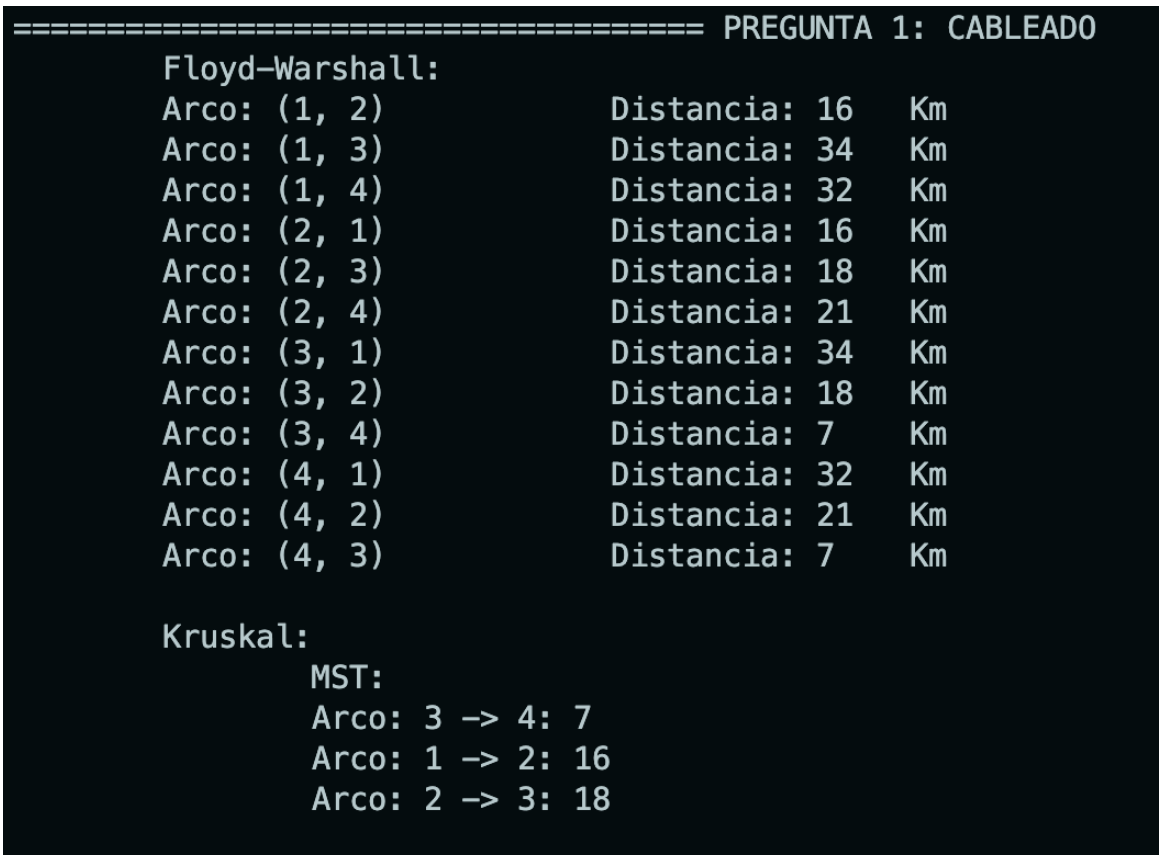
Kruskal

$O(n^2 * m)$

```cpp
//-----------------------------------------------------------------------------------------------
KRUSKAL'S ALGORITHM

// FUNCIÓN COMPAREWEIGHT: Función para comparar los pesos de dos aristas
// Complejidad --> O(1)
bool Graph::compareWeight(Edge* a, Edge* b){
    return a->weight < b->weight;
}

// FUNCIÓN KRUSKAL: Función para encontrar el árbol de expansión mínima
// Compljidad --> O(n^2 * m)
void Graph::runKruskal() {

    DisjointSets ds;
    // F = {}
    vector<Edge*> F;
    vector<Edge*>::iterator ei;
    // For each vertex v in G.V do MAKE-SET(v)
    for(ei = edges.begin(); ei != edges.end(); ++ei) {
        ds.makeSet((*ei)->first);
        ds.makeSet((*ei)->second);
    }

    // sort the edges of G.E by weight
    sort(edges.begin(), edges.end(), compareWeight);

    // For each edge (u, v) in G.E
    for(ei = edges.begin(); ei != edges.end(); ++ei) {
        vector<Node*> set1 = ds.findSet((*ei)->first);
        vector<Node*> set2 = ds.findSet((*ei)->second);
        // If FIND-SET(u) != FIND-SET(v)
        if(set1 != set2) {
            // F = F U {(u, v)} U {(v, u)}
            F.push_back((*ei));
            ds.do_union(set1, set2);
        }
    }

    // Return F
    cout << "MST: " << endl;
    vector<Edge*>::iterator ki;
    for(ki = F.begin(); ki != F.end(); ++ki) {
        cout << "\tArco: " <<(*ki)->first->number << " -> " << (*ki)->second->number << ": " << (*ki)-
>weight << endl;
    }

}
```

Resultados obtenidos con

Floyd-Warshall y Kruskal



Como se puede apreciar, en ambas imágenes el resultado obtenido es el mismo.

2) Ruta a seguir por el personal que reparte correspondencia

La segunda pregunta solicita marcar la ruta más barata que debería seguir el personal para recorrer todas las colonias y repartir la correspondencia. Dicha ruta inicia en un nodo, recorre los demás (1 sola vez) y, finalmente, debe regresar al nodo de partida. Este problema, trasladado a otro contexto similar, es conocido como Travelling Salesman Person (TSP).

TSP es un problema ampliamente referido en diversas aplicaciones tecnológicas. Además, es considerado un problema NP-Hard, es decir, como tal, no existe una solución que sea completamente eficiente y, además, la complejidad de su solución es directamente proporcional a la cantidad de destinaciones que se desee conocer (más ciudades representa mayor complejidad).

Debido a que existen diversas maneras para resolver este dilema de la programación, no resulta extraño encontrar soluciones con diversas complejidades asintóticas, por ejemplo, si nuestro enfoque es avaro (realizar todas las permutaciones del grafo, exhaustive search), el resultado da una complejidad de $O(n!)$.

Nuestra solución propuesta, la cuál se busca que sea lo más eficiente posible, se basa en el algoritmo Held-Karp. Consiste en generar varios subconjuntos que representen los diferentes caminos (y "subcaminos") podemos seguir. Cabe destacar que se considera un conjunto vacío y que el tamaño de los subconjuntos puede variar (de 0, que representa el conjunto vacío, hasta n-1, en donde n representa el número de nodos del grafo).

Teniendo en cuenta los subconjuntos resultantes, empezando por los más pequeños, buscaremos el camino más barato y lo almacenaremos. Posteriormente, para calcular el valor de los subconjuntos más grandes, nos apoyaremos de los resultados previamente almacenados (programación dinámica) con el fin de no sobrecargar al programa.

Cuando tenemos un subconjunto con un tamaño mayor a 1, significa que estamos considerando más de un camino para poder llegar a determinado nodo.

Debido a que se espera que solo 1 camino sea el más eficiente, es necesario almacenar, no solamente el mejor costo del camino, sino también el nodo seleccionado como la opción más barata. Por ello, para la implementación de este algoritmo tendremos 2 funciones: FunctionG (almacena el costo) y FunctionP (almacena el nodo predecesor).

Ambas funciones reciben los mismos parámetros: nodo destino y subconjunto que representa el camino.

Código

Held-Karp

$$O(n^2 * 2^n)$$

```
vector<int> HeldKarp::findHamilton(Node *start)
{
    // Llama a findHamilton(start, size_i) donde size_i va de 0 a N-1
    // Guarda functionP (obtiene el mejor costo) para el tamaño size_i y agrega a un vector de valores
P
    // Construye el path final usando el vector de valores P

    for (int s = 0; s < g->nodes.size() - 1; ++s) {
        // cout << "s: " << s << endl;
        findHamilton(start, s);
    }

    vector<int> finalSet;
    // Agrega todos los nodos a un set
    vector<Node *>::iterator ni;
    for (ni = g->nodes.begin(); ni != g->nodes.end(); ++ni) {
        finalSet.push_back((*ni)->number);
    }
    // Quita el nodo inicial
    finalSet = values_without(finalSet, start->number);

    FunctionG *fg = new FunctionG(start->number, finalSet);
    fg->result = findResultG(fg);

    vector<FunctionG *> finalLevel;
    finalLevel.push_back(fg);
    prev_results.push_back(finalLevel);

    // Opcional. Para comprobar el funcionamiento:
    cout << "Functions G: {" << endl;
    vector<vector<FunctionG *>>::iterator ri;
    for (ri = prev_results.begin(); ri != prev_results.end(); ++ri) {
        cout << endl << "[" << endl;
        vector<FunctionG *>::iterator fi;
        for (fi = (*ri).begin(); fi != (*ri).end(); ++fi) {
            cout << (*fi)->toString() << endl;
        }
        cout << "]" << endl;
    }
    cout << "}" << endl;

    // Encontrar los valores P
    vector<int> valuesP = findResultP(start);
    // for (ri = prev_results.begin(); ri != prev_results.end(); ++ri) {
    //     int p = findResultP(*ri, start);
    //     valuesP.push_back(p);
    // }

    // valuesP.push_back(start->number); // Salir del inicio

    // Opcional. Mostrar el recorrido TSP obtenido
    cout << endl << "\tTSP: ";
    vector<int>::reverse_iterator pi;
    for (pi = valuesP.rbegin(); pi != valuesP.rend(); ++pi) {
        if (pi + 1 != valuesP.rend())
            cout << to_string(*pi) << " --> ";
        else
            cout << to_string(*pi) << endl;
    }
    return valuesP;
}
```

```cpp
// FUNCIÓN FINDHAMILTON
// Complejidad: O(n^2)
// Construye todos los subconjuntos de tamaño set_size
void HeldKarp::findHamilton(Node *start, int set_size) {
    if (set_size == 0) {
        vector<FunctionG *> prev; // prev es nuevo porque estamos en tamaño 0.

        // Tomar los nodos que llegan a start y construir conjuntos + FunctionG.
        vector<Edge *>::iterator ei;
        for (ei = g->edges.begin(); ei != g->edges.end(); ++ei) {
            if ((*ei)->second == start){ // Llega a start
                vector<int> vi; // conjunto de tamaño 0
                FunctionG *fg = new FunctionG((*ei)->first->number, vi); // FunctionG con el nodo que
llega a start
                fg->result = (*ei)->weight; // Costo de la arista
                // Lo anterior es:   g(vecino de start, {})
                prev.push_back(fg);
            }
        }
        prev_results.push_back(prev);
    } else {
        vector<FunctionG *> curr;
        vector<FunctionG *> prev = prev_results[set_size - 1]; // prev es el vector anterior.

        // Ahora, los conjuntos salen de la combinación de la iteración anterior.
        vector<FunctionG *>::iterator gi;
        vector<int> values;

        // A. Toma los valores (únicos) de salida anteriores.
        for (gi = prev.begin(); gi != prev.end(); ++gi) {
            // Agrega los valores de salida a un vector
            if (std::find(values.begin(), values.end(), (*gi)->exit_val) == values.end()) { // Si no se
encuentra
                values.push_back((*gi)->exit_val); // Agregarlo para que sea único
            }
            // cout << "values: " << (*gi)->exit_val << endl;
        }

        // B. Combínalos con conjuntos creados a partir de los demás valores, sin pasar del tamaño del
conjunto.
            //Si es necesario, hacer más conjuntos.
        vector<int>::iterator vi;
        for (vi = values.begin(); vi != values.end(); ++vi) {
            vector<int> superset = values_without(values, *vi); // Quita el valor actual
            vector<vector<int>> subsets; // Subconjuntos de tamaño set_size
            vector<int> d(set_size, 0); // Vector de ceros tamaño set_size
            findCombinations(superset, superset.size(), set_size, 0, d, 0, subsets);
            vector<vector<int>>::iterator si;
            for (si = subsets.begin(); si != subsets.end(); ++si) {
                FunctionG *fg = new FunctionG(*vi, *si);
                fg->result = findResultG(fg);
                curr.push_back(fg);
            }
        }
        prev_results.push_back(curr);
    }
}
```

```cpp
// FUNCIÓN VALUES_WITHOUT: Regresa un vector de enteros con todos los valores excepto "to_remove"
// Complejidad: O(n)
vector<int> HeldKarp::values_without(vector<int> &v, int to_remove){
    vector<int> result;
    vector<int>::iterator vi;
    for (vi = v.begin(); vi != v.end(); ++vi)
        if ((*vi) != to_remove) result.push_back(*vi);
    return result;
}

/*
FUNCIÓN FINDCOMBINATIONS:
    Encuentra los posibles subconjuntos de arr (arr de tamaño n).
    Almacena estos subconjuntos en "combos".
    Los subconjuntos son de tamaño r.
    Original de: https://www.geeksforgeeks.org/print-subsets-given-size-set/

    Complejidad: O(n^r)
*/
void HeldKarp::findCombinations(vector<int> &arr, int n, int r, int index, vector<int> &data, int i,
vector<vector<int>> &combos) {
    if (index == r) {
        // Un subconjunto está listo (de data[0] a data[r]), lo guardamos en combos:
        vector<int> subset;

        for (int j = 0; j < r; j++){
            subset.push_back(data[j]);
        }
        combos.push_back(subset);
        return;
    }

    if (i >= n) return; // No hay más elementos
    data[index] = arr[i]; // Agrega el elemento a data
    findCombinations(arr, n, r, index + 1, data, i + 1, combos); // Recursión
    findCombinations(arr, n, r, index, data, i + 1, combos); // Recursión
}
```

```cpp
/*
FUNCIÖN COMPARESETS:
    Compara dos vectores de enteros.
    Regresa true si contienen los mismos elementos (no necesariamente ordenados), false si no.
    Complejidad: O(n^2)
*/
bool HeldKarp::compareSets(vector<int> a, vector<int> b) {
    if (a.size() == b.size()){
        vector<int>::iterator ia;
        for (ia = a.begin(); ia != a.end(); ++ia){
            if (std::find(b.begin(), b.end(), *ia) == b.end()) return false;
        }
        return true;
    }
    return false;
}

/*
FUNCIÓN findResultG:
    Busca en prev_results el resultado de la función g(e, s) para un conjunto s.
    Si no encuentra dicha función, regresa -1.
    Complejidad: O(n^2)
*/
int HeldKarp::findPrevValueG(int e, vector<int> s)
{
    int sizeToFind = s.size();
    vector<FunctionG *> search_space = prev_results[sizeToFind];
    vector<FunctionG *>::iterator gi;
    for (gi = search_space.begin(); gi != search_space.end(); ++gi)
    {
        if ((*gi)->exit_val == e && compareSets(s, (*gi)->set))
            return (*gi)->result;
    }

    FunctionG *fg = new FunctionG(e, s);
    cout << "Warning: Previous FuncionG not found: " << fg->toString() << endl;
    return -1;
}

/*
FUNCIÓN findResultG:
    Regresa el resultado de la función g(e, s).
    g(exit, S) = min{Edge(Si, exit) + g(Si, S - {Si})} para toda Si en el conjunto S.
    Es decir, g(exit, S) = min{peso del arco de Si hasta exit + g(Si, S menos el elemento Si)}.
    Complejidad: O(n^2)
*/
int HeldKarp::findResultG(FunctionG *fg) {
    vector<int>::iterator si;
    vector<int> candidates;

    for (si = fg->set.begin(); si != fg->set.end(); ++si) {
        vector<int> subset = values_without(fg->set, *si);
        Edge *e = g->findEdge2(fg->exit_val, *si);

        if (e != nullptr){
            int c = e->weight + findPrevValueG(*si, subset);
            candidates.push_back(c); // weight(Si, exit) + g(Si, S - {Si})
        } else {
            cout << "Warning: Edge not found from: " << to_string(fg->exit_val) << "to" <<
to_string(*si) << "." << endl;
        }
    }

        if (candidates.size() == 0) {
            cout << "Warning: candidates for findWeight is empty: " << fg->toString() << endl;
            return -1;
        }

        int minC = *std::min_element(candidates.begin(), candidates.end());
        return minC;
}
```

```cpp
/*
FUNCIÓN findResultP:
    Para encontrar el valor de P:
        1. Encontrar el mínimo valor de fg para este tamaño de conjunto.
        2. El valor de P será el penúltimo nodo en el conjunto S, hacia la salida.
        2.1 Si S == {} entonces P = salida.
        2.2 Si |S| < 2 entonces no tiene penúltimo elemento y se toma el único disponible.

    Complejidad: O(n^2)
*/
vector<int> HeldKarp::findResultP(Node* start)
{
    vector<int> tsp = { start->number };
    vector<int> tempSet;
    int counter = 1;
    for (int r = prev_results.size() - 1; r >= 0; r--)
    {
        vector<FunctionG*> gs = prev_results[r];
        if (r == prev_results.size() - 1)
        {
            int step = -1;
            if (gs[0]->set.size() > 1) step = gs[0]->set[1];
            else step = gs[0]->set[0];
            tempSet = values_without(gs[0]->set, step);
            tsp.push_back(step);
        }
        else
        {
            int step = -1;
            vector<FunctionG*>::iterator gi;
            for (gi = gs.begin(); gi != gs.end(); ++gi)
            {
                if ((*gi)->exit_val == tsp[counter-1] && compareSets(tempSet, (*gi)->set))
                {
                    if ((*gi)->set.size() > 1) step = (*gi)->set[1];
                    else if ((*gi)->set.size() == 1) step = (*gi)->set[0];
                    else step = start->number;

                    tempSet = values_without((*gi)->set, step);
                    tsp.push_back(step);
                }
            }
        }
        counter++;
    }
    return tsp;
}
```

```cpp
/*
FUNCIÓN findResultG:
    Busca en prev_results el resultado de la función g(e, s) para un conjunto
s.  Si no encuentra dicha función, regresa -1.
    Complejidad: O(n^2)
*/
int HeldKarp::findPrevValueG(int e, vector<int> s)
{
    int sizeToFind = s.size();
    vector<FunctionG *> search_space = prev_results[sizeToFind];
    vector<FunctionG *>::iterator gi;
    for (gi = search_space.begin(); gi != search_space.end(); ++gi)
    {
        if ((*gi)->exit_val == e && compareSets(s, (*gi)->set))
            return (*gi)->result;
    }

    FunctionG *fg = new FunctionG(e, s);
    cout << "Warning: Previous FuncionG not found: " << fg->toString() << endl;
    return -1;
}
```

Resultados obtenidos con Held-Karp

```
================================== PREGUNTA 2: MEJOR RUTA
El camino más corto es:
Functions G: {

[
G(2, {}) = 16
G(4, {}) = 32
G(3, {}) = 45
]

[
G(2, {4, }) = 53
G(2, {3, }) = 63
G(4, {2, }) = 37
G(4, {3, }) = 52
G(3, {2, }) = 34
G(3, {4, }) = 39
]

[
G(2, {4, 3, }) = 57
G(4, {2, 3, }) = 41
G(3, {2, 4, }) = 44
]

[
G(1, {2, 3, 4, }) = 73
]
}

        TSP: 1 --> 2 --> 4 --> 3 --> 1
```

3) *Valor de flujo máximo de información*

Para determinar el Flujo máximo de información se empleó el algoritmo de Ford-Fulkerson. Con este método es posible determinar el flujo máximo proporcionando 2 nodos: origen y sumidero. El algoritmo plantea un grafo cuyas aristas poseen un flujo determinado, por lo que el flujo entrante debe ser el mismo que el saliente y no debe exceder a la capacidad máxima de la arista dada. En la implementación realizada se hace un recorrido de tipo BFS (*Breadth-First Search*) que utiliza el método FIFO (First In, First Out) para encontrar el camino más corto de un grafo. La complejidad calculada para este algoritmo es de $O(n^3 * m)$.

Código

Ford-Fulkerson

$O(n^3 * m)$

```
//-------------------------------------FORD-FULKERSON
//FUNCIÓN FORD-FULKERSON: Calcula el flujo máximo en un grafo
//Complejidad --> O(n^3 * m)
int Graph::runFordFulkerson(int source, int sink) {
    Node* s = findNode(source);
    Node* t = findNode(sink);

    if (s == nullptr || t == nullptr) return -100000;

    int maxFlow = 0;
    for (Edge* e : edges) {
        // Actualizar el flujo (flow) de cada arco con el valor de capacidad (weight)
        e->flow = e->weight;
        while(bfs(s, t)) {
            int pathFlow = INT_MAX;
            // Nodo temporal para recorrer el path
            Node *curr = t;
            while (curr != s) {
                // Obtener la arista que conecta el nodo previo con el nodo actual
                Edge *e = findEdge(curr->prev, curr);
                // Encontrar el flujo mínimo
                pathFlow = min(pathFlow, e->flow);
                curr = curr->prev;
            }
            // Se realiza el camino inverso para actualizar los flujos
            curr = t;
            // Mientras el nodo actual no sea el nodo s
            while(curr != s) {
                // Buscar arista que conecta el nodo previo con el nodo actual
                //para actualizar el flujo
                Edge *e1 = findEdge(curr->prev, curr);
                e1->flow -= pathFlow;
```

```
                // Buscar arista que conecta el nodo actual con el nodo previo para
                //actualizar el flujo
                Edge *e2 = findEdge(curr, curr->prev);
                if (e2 != nullptr) e2->flow += pathFlow;
                curr = curr->prev;
            }
            // Actualizar el flujo máximo
            maxFlow += pathFlow;
        }
    }
    return maxFlow;
}
```

Resultados obtenidos con Ford-Fulkerson

```
==================================== PREGUNTA 3: FLUJO MÁXIMO
        Flujo Máximo de Información: 78
```

4) Búsqueda geométrica dada una nueva contratación

En la última pregunta nos solicitan determinar la central más cercana a la que se puede conectar una nueva casa (representa una nueva contratación).

Recibimos una lista de coordenadas en el formato (x, y), las cuales representan la posición geográfica de todas las centrales disponibles. Cabe destacar que el último punto (último registro en la lista) hace referencia a la posición geográfica de la casa que busca conexión.

En aras de resolver esta pregunta, usaremos la fórmula matemática para calcular la distancia entre 2 puntos:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2};$$

En donde (x_1, y_1) representa las coordenadas de un punto, y (x_2, y_2) de otro.

Para encontrar la central con la menor distancia a la nueva contratación, uno de los puntos en la ecuación tendrá los valores de las coordenadas de la casa y, el otro punto, tendrá los de las centrales (usaremos un ciclo for para ir cambiándolo).

Código

Búsqueda Geométrica

$O(n * log(n) \ + n)$

```cpp
//FUNCIÓN CALCULA-DISTANCIA-MÍNIMA: Calcula la menor distancia entre
//todos los puntos
//Complejidad: O(n*log(n) + n)
void GSearch::calculaDistMinima2(vector<Point> puntos){
    vector<Point> comp(puntos.begin(), puntos.end() - 1);
    Point casa = puntos[puntos.size() - 1];
    vector<pair<Point, float>> dists;

    //Complejidad: O(n)
    for (int i = 0; i < comp.size(); i++){
        float dist = sqrt(pow(comp[i].x - casa.x, 2) + pow(comp[i].y - casa.y, 2));
        dists.push_back({comp[i], dist});
    }

    //Complejidad: O(n*log(n))
    sort(puntos.begin(), puntos.end(), sortByX);
    cout << "\tPuntos ordenados de izquierda a derecha: " << endl;
    int index = 1;
    //Complejidad: O(n)
    for (Point p : puntos){
        cout << "\t" << index++ << ".- (" << p.x  << ", " << p.y << ")" << endl;
    }
    cout << endl;

    //Complejidad: O(n)
    for (int i = 0; i < comp.size(); i++){
        cout << "\tDistancia de la nueva contratación (" << casa.x << ", " << casa.y << ") a la tienda
(" << comp[i].x << ", " << comp[i].y << "): " << dists[i].second << "Km" << endl;
    }

    //Complejidad: O(n*log(n))
    sort(dists.begin(), dists.end(), sortPairByDist);
    cout << "\n\tLa central más cercana a la nueva contratación es: (" << dists[0].first.x << ", " <<
dists[0].first.y << ") con una distancia de " << dists[0].second << "Km" << endl;
}
```

Resultados obtenidos con
Búsqueda Geométrica

```
================================== PREGUNTA 4: LISTA DE SUCURSALES
        Puntos ordenados de izquierda a derecha:
        1.- (200, 500)
        2.- (300, 100)
        3.- (301, 101)
        4.- (450, 150)
        5.- (520, 480)

        Distancia de la nueva contratación (301, 101) a la tienda (200, 500): 411.585Km
        Distancia de la nueva contratación (301, 101) a la tienda (300, 100): 1.41421Km
        Distancia de la nueva contratación (301, 101) a la tienda (450, 150): 156.85Km
        Distancia de la nueva contratación (301, 101) a la tienda (520, 480): 437.724Km

        La central más cercana a la nueva contratación es: (300, 100) con una distancia de 1.41421Km
```

Conclusiones

Hoy en día, contamos con diversas herramientas computacionales que nos permiten resolver problemas de la vida cotidiana con mayor facilidad. Sin duda, es relevante para un profesional en el área, conocer y comprender diferentes algoritmos que existen para poder ponerlos en práctica en el mundo real y brindar soluciones que sean óptimas y que hagan un uso eficiente de los recursos con los que se cuenta.

En lo personal, esta actividad me resultó bastante interesante ya que pude reflexionar sobre la utilidad que tienen los diferentes algoritmos implementados en las últimas semanas con respecto a teoría de grafos y geometría computacional para estudiar, modelar y resolver problemas. De igual forma, me dio la oportunidad de darme cuenta de que, si bien se puede resolver un problema aplicando diferentes algoritmos y llegar a una solución que sea válida, es fundamental identificar los algoritmos que mejor se adapten a la situación en cuestión, ya que cada uno de ellos puede tener un rendimiento distinto por factores que intervienen en la problemática, como la cantidad de datos a representar en nodos o el número de conexiones entre ellos, por ejemplo.

Referencias

GeeksforGeeks.(2022b, octubre 16).*Kruskal's Minimum Spanning Tree Algorithm | Greedy Algo-2*. https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

GeeksforGeeks. (2022, noviembre 11). *Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming)*. https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

GeeksforGeeks. (2022, agosto 31). *Dijsktra's algorithm*. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

GeeksforGeeks. (2022c, agosto 31). *Floyd Warshall Algorithm | DP-16*. https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

GeeksforGeeks. (2022a, junio 21). *Ford-Fulkerson Algorithm for Maximum Flow Problem*. https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/

GeeksforGeeks. (2020, 4 junio). *Proof that traveling salesman problem is NP Hard*. https://www.geeksforgeeks.org/proof-that-traveling-salesman-problem-is-np-hard/