|

AI1_Actividad Integradora 2

Alumnos:

Luis Gabriel Delfín Paulín | A01701482
Jorge Emiliano Pomar Mendoza | A01709338
Eliuth Balderas Neri | A01703315

Análisis y diseño de algoritmos avanzados (Gpo 601)

Profesor: Ramona Fuentes Valdéz

Tecnológico de Monterrey, campus Querétaro

Febrero- Junio 2024

1 de mayo del 2024

Índice

Introducción a la Problemática

El acceso a internet en México sigue siendo un lujo y privilegio para la mayoría de la población. Tanto por la desigualdad que se vive en las zonas rurales como en las ciudades grandes, actualmente hay muchas zonas que quedan fuera de una cobertura de la red de telecomunicaciones. Aunque en los últimos años, específicamente en el sexenio de 2012-2018 que se aprobó la concesión del gobierno a la empresa Altán para llevar el proyecto de la red compartida en México, aún siguen existiendo muchos problemas de cobertura.

Yéndonos un poco al pasado, en 2020. Año en el que inició el confinamiento de la pandemia. El rezago educativo se fue a la alza debido al cierre de las escuelas y la falta de infraestructura tecnológica de comunicación para que todos los estudiantes en México pudieran conectarse a clases virtuales. Cuatro años después, la pandemia ya terminó. De todas maneras, sigue existiendo la misma desigualdad en cuanto al acceso a internet tanto para la educación como para la comunicación. Es por eso que en esta situación problema nos hacemos las siguientes preguntas con el fin de encontrar una solución a la problemática presentada.

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

Descripción

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.

Implementación

Para encontrar la mejor forma de cablear la fibra óptica en las colonias, implementamos el algoritmo de Kruskal. Dicho algoritmo nos ayudó a determinar el árbol de expansión mínima (MST).

Cabe resaltar que utilizamos la estructura de Union Find para preparar el algoritmo que encuentra el spanning tree.

Algoritmo de Kruskal

```
// Para Kruskal primero se ordenan las aristas por peso
bool compEdge(const Edge &e1, const Edge &e2) { return e1.weight <
e2.weight; }

// Función para imprimir el grafo min spanning tree
void printGraph(std::vector<Edge> &spanningTree) {
  for (auto &e : spanningTree) {
    std::cout << char('A' + e.src) << " - " << char('A' + e.dest) <<
" ("
              << e.weight << " km)" << std::endl;
  }
}
```

```
// Kruskal para encontrar el MST arbol de expansion minima del grafo
std::vector<Edge> kruskalMST(Graph &graph) {
  std::vector<Edge> mst;
  sort(graph.edges.begin(), graph.edges.end(), compEdge);
  UnionFind uf(graph.V);

  for (auto &edge : graph.edges) {
    if (uf.find(edge.src) != uf.find(edge.dest)) {
      uf.unionSet(edge.src, edge.dest);
      mst.push_back(edge);
    }
  }
  return mst;
}
```

Algoritmo de Prim

```
// Algoritmo de Prim para encontrar el MST (árbol de expansión
mínima) del grafo
// basado en
https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-
algo-5/
void primMST(Graph& graph) {
  int V = graph.V;
  std::vector<int> parent(V);
  std::vector<int> key(V, INT_MAX);
  MinHeap minHeap(V);

  for (int v = 0; v < V; ++v) {
    minHeap.array[v] = {v, key[v]};
    minHeap.pos[v] = v;
  }

  minHeap.pos[0] = 0;
  key[0] = 0;
  minHeap.array[0] = {0, key[0]};
  minHeap.size = V;
```

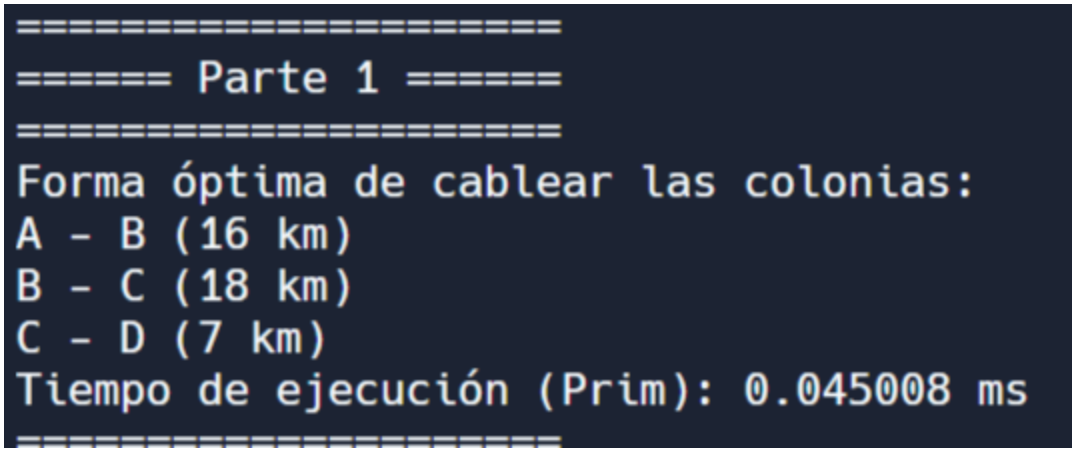
```
  parent[0] = -1;

  while (minHeap.size > 0) {
    MinHeapNode minHeapNode = minHeap.extractMin();
    int u = minHeapNode.v;

    for (int v = 0; v < V; ++v) {
      if (graph.adjMatrix[u][v] && minHeap.isInMinHeap(v) &&
graph.adjMatrix[u][v] < key[v]) {
        key[v] = graph.adjMatrix[u][v];
        parent[v] = u;
        minHeap.decreaseKey(v, key[v]);
      }
    }
  }
  printArr(parent, graph.adjMatrix, V);
}
```

Resultados

```
======================
====== Parte 1 ======
======================
Forma óptima de cablear las colonias:
C - D (7 km)
A - B (16 km)
B - C (18 km)
Tiempo de ejecución (Kruskal): 0.0029 ms
```



Análisis de Complejidad / Justificación

El algoritmo de **Kruskal** es un método en la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Busca un subconjunto de aristas que forman un árbol incluyendo todos los vértices, con la suma total mínima de las aristas. Si el grafo no es conexo, encuentra un bosque expandido mínimo (un árbol recubridor mínimo para cada componente conexo). Nombrado en honor a Joseph Kruskal, quien lo publicó en 1956, este algoritmo es un ejemplo de algoritmo voraz. Otros algoritmos para hallar el árbol de expansión mínima incluyen el algoritmo de Prim, el algoritmo del borrador inverso y el algoritmo de Boruvka.
El algoritmo de Kruskal procede de la siguiente manera:
Se crea un bosque B (un conjunto de árboles), donde cada vértice del grafo es un árbol separado.
Se crea un conjunto C que contenga todas las aristas del grafo.
Mientras C no esté vacío:
Se elimina una arista de peso mínimo de C.
Si esa arista conecta dos árboles diferentes, se añade al bosque, combinando los dos árboles en uno solo.
En caso contrario, se desecha la arista.
Al finalizar el algoritmo, el bosque tiene un solo componente, formando un árbol de expansión mínimo del grafo. En un árbol de expansión mínima, la cantidad de aristas es igual a la cantidad de nodos menos uno (1).

Por otra parte, el algoritmo de **Prim** es un método en la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo, no dirigido y con aristas etiquetadas. Este algoritmo identifica un subconjunto de aristas que forman un árbol incluyendo todos los vértices, donde el peso total de las aristas es el menor posible. Si el grafo no es conexo, el algoritmo encuentra el árbol recubridor mínimo para uno de los componentes conexos. Fue diseñado en 1930 por el matemático Vojtech Jarnik y redescubierto de manera independiente por Robert C. Prim en 1957 y por Dijkstra en 1959, por lo que también se le conoce como algoritmo DJP o algoritmo de Jarnik.

El algoritmo incrementa gradualmente el tamaño del árbol comenzando con un vértice inicial y agregando sucesivamente los vértices conectados por la arista de menor peso disponible. En cada paso, se consideran las aristas que conectan vértices dentro del árbol con vértices fuera del árbol. El proceso continúa hasta que todos los vértices se han incluido en el árbol recubridor mínimo.

Los pasos del algoritmo son los siguientes:

Inicializar un árbol con un único vértice, elegido arbitrariamente.
Expandir el árbol seleccionando la arista de menor peso que conecte el árbol con un vértice aún no incluido.
Repetir el paso 2 hasta incluir todos los vértices en el árbol.

Complejidad Temporal Kruskal

Primero para ordenar las aristas, usamos un algoritmo el cual tiene la complejidad de **O (E log E)** donde E representa el número de aristas o conexiones.

Después, cada una de las listas ya procesadas pasan a un proceso de búsqueda y unión con el algoritmo de Kruskal. En este proceso, la complejidad es de **O(Eα(V)).**
Es por esto que la complejidad total de la implementación de esta parte 1 es la suma de los dos procesos. **O(ElogE)+O(Eα(V))**

Complejidad Temporal Prim

En esta implementación se usa primero un min Heap que inicializa los nodos V, tiene una complejidad de **O(V)**. Dentro de la misma min Heap está la función de extractMinHeap que tiene una complejidad de **O(log V)**. Mismo caso para la función decreaseKey. Después se exploran los vértices adyacentes de cada nodo, dando una complejidad de **O(E log V)**. Así que por último al juntar las dos operaciones principales del algoritmo tenemos que O(V log V) + O(E log V). Dando una simplificación de **O((V+E) log V)**.

Parte 2

Descripción

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.

Implementación

El algoritmo del **Vecino Más Cercano** se utilizó en el código para resolver una versión aproximada del TSP debido a su simplicidad, velocidad y capacidad para proporcionar soluciones razonablemente buenas en tiempo rápido.

El algoritmo de **Ramificación y Acotamiento** se utilizó en el código para resolver el problema de flujo máximo entre dos colonias en un grafo ponderado debido a su capacidad para encontrar la solución óptima garantizada y su adaptabilidad a problemas de redes complejas.

```
// TSP con el algoritmo del vecino más cercano (Nearest Neighbor)
para la ruta
// más corta que visita cada colonia exactamente 1 vez
std::vector<int> tspNearestNeighbor(const Graph &graph, int start) {
  int n = graph.V;
  std::vector<bool> visited(n, false);
  std::vector<int> path;
  int current = start;
  path.push_back(current);
  visited[current] = true;
  int total_distance = 0;
```

```
  for (int i = 0; i < n - 1; ++i) {
    int nearest = -1;
    int nearest_dist = std::numeric_limits<int>::max();
    for (int j = 0; j < n; ++j) {
      if (!visited[j] && graph.edges[current * n + j].weight > 0 &&
          graph.edges[current * n + j].weight < nearest_dist) {
        nearest_dist = graph.edges[current * n + j].weight;
        nearest = j;
      }
    }
    if (nearest != -1) {
      path.push_back(nearest);
      visited[nearest] = true;
      total_distance += nearest_dist;
      current = nearest;
    }
  }
  total_distance += graph.edges[current * n + start].weight;
  path.push_back(start);
  std::cout << "Ruta del Viajero:" << std::endl;
  for (int i : path) {
    std::cout << char('A' + i) << " ";
  }
  std::cout << "\nDistancia total: " << total_distance << " km" <<
std::endl;

  return path;
}
```

Análisis de Complejidad / Justificación

Nearest Neighbor

El algoritmo del **Vecino Más Cercano** es una heurística para resolver el Problema del Viajero (TSP), el cuál consiste en encontrar una ruta que visite cada ciudad exactamente una vez y regrese al punto de partida, minimizando la distancia total recorrida. Este algoritmo no garantiza encontrar la solución óptima, pero proporciona una solución rápida y sencilla, teniendo un enfoque voraz.

El algoritmo comienza en una ciudad inicial y, en cada paso, se mueve a la ciudad no visitada más cercana. Este proceso se repite hasta que se hayan visitado todas las ciudades, y luego se regresa a la ciudad de inicio para completar el ciclo. La elección de la ciudad más cercana en cada paso es lo que caracteriza al algoritmo como una heurística voraz, ya que toma decisiones óptimas en cada etapa sin considerar el impacto global.
Los pasos del algoritmo pueden describirse así:

- Seleccionar una ciudad inicial.
- Desde la ciudad actual, mover a la ciudad no visitada más cercana.
- Marcar la ciudad actual como visitada.
- Repetir los pasos 2 y 3 hasta que todas las ciudades hayan sido visitadas.
- Regresar a la ciudad inicial para completar la ruta.

Complejidad Temporal Nearest Neighbor

Esta implementación primero se tiene un vector de ciudades visitadas que tiene complejidad de O(N) que representa el número de vértices que se interpretan como las ciudades. Después hay un bucle que itera sobre todos los vértices para checar si ya fue visitado o no, es decir se ejecuta n - 1 de veces. Así que la complejidad principal es multiplicar el vector y el bucle **O(n)×(n−1)** que da como resultado **O(**N^2**).**

Branch and Bound

Por otra parte, el algoritmo de **Ramificación y Acotamiento** (Branch and Bound) es un método exacto para resolver el Problema del Viajero (TSP), que consiste en encontrar la ruta más corta que visite cada ciudad exactamente una vez y regrese al punto de partida. Este algoritmo busca la solución óptima explorando todas las posibles rutas, pero utiliza estrategias de poda para descartar aquellas ramas del árbol de búsqueda que no pueden mejorar la solución actual.
El algoritmo de Ramificación y Acotamiento explora todas las posibles rutas mediante un árbol de búsqueda. Cada nodo del árbol representa una ruta parcial, y se calcula una cota inferior para la longitud total de la ruta que pasa por ese nodo. Si la cota inferior de una ruta parcial excede la longitud de la mejor ruta encontrada hasta el momento, esa ruta se descarta. Este enfoque permite reducir el número de rutas a considerar, optimizando la búsqueda de la mejor solución.

Los pasos del algoritmo pueden describirse informalmente como sigue:

- Inicializar la mejor solución como infinita y crear un nodo raíz con la ciudad inicial y una cota inferior calculada.
- Insertar el nodo raíz en una cola de prioridad, ordenada por cota inferior.

- Mientras la cola de prioridad no esté vacía:
    - Extraer el nodo con la menor cota inferior.
    - Si la cota inferior del nodo es mayor o igual a la mejor solución encontrada, descartar el nodo.
    - Si el nodo representa una solución completa (todas las ciudades visitadas y regreso a la ciudad inicial):
        - Actualizar la mejor solución si la nueva ruta es más corta.
    - Para cada ciudad no visitada, crear un nodo hijo expandiendo la ruta parcial e insertar el hijo en la cola de prioridad.
- La mejor solución encontrada al finalizar el proceso es la ruta óptima.

Complejidad Temporal Branch & Bound

Esta implementación tiene muchas partes que contribuyen a la complejidad temporal que hace eficiente este algoritmo. En primer lugar, para encontrar el camino de los nodos tiene una complejidad de O(N) donde N que es el tamaño del camino que puede variar. Después, se tiene la búsqueda del mínimo costo donde se recorren todos los nodos N anteriores hasta encontrar al mejor, por lo que lo hace tener una complejidad de $\mathbf{O(}N^2\mathbf{)}$. Hasta ahí llevamos $\mathbf{O(}N^3\mathbf{)}$**.** Después, la función de branchAndBoundTSP tiene una priority queue donde se van insertando cada entrada. Además tiene operaciones como top y pop. Cada inserción y cada operación tiene complejidad de O(log K) donde K representa el número de nodos que existen en la priority queue. Y además por eso se podría decir que el peor de los casos donde la función no hace ninguna poda, el algoritmo tendría que hacer N! de inserciones.

Juntando las dos operaciones, tenemos que la complejidad total es de $O(N\mathbf{!}) \times O(N^3\mathbf{)}$

Implementación de Vecino más Cercano

```
// TSP con el algoritmo del vecino más cercano (Nearest Neighbor)
para la ruta
// más corta que visita cada colonia exactamente 1 vez
std::vector<int> tspNearestNeighbor(const Graph &graph, int start) {
  int n = graph.V;
  std::vector<bool> visited(n, false);
  std::vector<int> path;
  int current = start;
```

```
  path.push_back(current);
  visited[current] = true;
  int total_distance = 0;

  for (int i = 0; i < n - 1; ++i) {
    int nearest = -1;
    int nearest_dist = std::numeric_limits<int>::max();
    for (int j = 0; j < n; ++j) {
      if (!visited[j] && graph.edges[current * n + j].weight > 0 &&
          graph.edges[current * n + j].weight < nearest_dist) {
        nearest_dist = graph.edges[current * n + j].weight;
        nearest = j;
      }
    }
    if (nearest != -1) {
      path.push_back(nearest);
      visited[nearest] = true;
      total_distance += nearest_dist;
      current = nearest;
    }
  }
  total_distance += graph.edges[current * n + start].weight;
  path.push_back(start);
  std::cout << "Ruta del Viajero:" << std::endl;
  for (int i : path) {
    std::cout << char('A' + i) << " ";
  }
  std::cout << "\nDistancia total: " << total_distance << " km" <<
std::endl;

  return path;
}
```

Implementación de Ramificación y Acotamiento (Branch and Bound)

```
// Función para calcular la cota inferior de un nodo en el algoritmo
de Ramificación y Acotamiento
```

```
int calculateBound(const TSPNode& node, const Graph& graph, int N) {
  int bound = node.cost;
  for (int i = 0; i < N; ++i) {
    if (std::find(node.path.begin(), node.path.end(), i) ==
node.path.end()) {
      int minCost = INT_MAX;
      for (int j = 0; j < N; ++j) {
        if (i != j && std::find(node.path.begin(), node.path.end(),
j) == node.path.end()) {
          minCost = std::min(minCost, graph.adjMatrix[i][j]);
        }
      }
      bound += minCost;
    }
  }
  return bound;
}

// Algoritmo de Ramificación y Acotamiento para resolver el problema
del viajero
void branchAndBoundTSP(const Graph& graph, int start) {
  int N = graph.V;
  std::priority_queue<TSPNode, std::vector<TSPNode>, CompareTSPNode>
pq;
  TSPNode root = {{start}, 0, 0};
  root.bound = calculateBound(root, graph, N);
  pq.push(root);

  TSPNode bestNode;
  bestNode.cost = INT_MAX;

  while (!pq.empty()) {
    TSPNode current = pq.top();
    pq.pop();

    if (current.bound < bestNode.cost) {
      for (int i = 0; i < N; ++i) {
        if (std::find(current.path.begin(), current.path.end(), i) ==
current.path.end()) {
```

```
          TSPNode child = current;
          child.path.push_back(i);
          child.cost += graph.adjMatrix[current.path.back()][i];
          if (child.path.size() == N) {
            child.path.push_back(start);
            child.cost += graph.adjMatrix[i][start];
            if (child.cost < bestNode.cost) {
              bestNode = child;
            }
          } else {
            child.bound = calculateBound(child, graph, N);
            if (child.bound < bestNode.cost) {
              pq.push(child);
            }
          }
        }
      }
    }
  }

  std::cout << "Ruta del Viajero:" << std::endl;
  for (int i : bestNode.path) {
    std::cout << char('A' + i) << " ";
  }
  std::cout << "\nDistancia total: " << bestNode.cost << " km" <<
std::endl;
}
```

Parte 3

Descripción

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.

Implementación

El algoritmo de **Edmonds-Karp** se utilizó en el código para encontrar el flujo máximo entre dos colonias en un grafo ponderado debido a su eficiencia garantizada, facilidad de implementación y capacidad para encontrar la solución óptima al problema de flujo máximo en la red.

Implementación de Edmons-Karp

```
std::vector<int> bfsEdmondsKarp(Graph &graph, int src, int dest,
                                std::vector<std::vector<int>>
&residualGraph) {
  std::vector<int> parent(graph.V, -1);
  std::vector<bool> visited(graph.V, false);

  std::queue<int> path;
  path.push(src);
  visited[src] = true;

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

    for (int v = 0; v < graph.V; v++) {
      if (!visited[v] && residualGraph[u][v] > 0) {
        parent[v] = u;
        visited[v] = true;
        path.push(v);
        if (v == dest)
          return parent;
      }
    }
```

```
  }

  return parent;
}

// Con el grafo de arriba se calcula el flujo máximo entre la colonia
0 y la
// colonia N-1
int edmondsKarp(Graph &graph, int src, int dest) {
  std::vector<std::vector<int>> residualGraph(graph.V,

std::vector<int>(graph.V, 0));
  for (auto &edge : graph.edges) {
    residualGraph[edge.src][edge.dest] = edge.weight;
  }

  int maxFlow = 0;
  while (true) {
    std::vector<int> parent = bfsEdmondsKarp(graph, src, dest,
residualGraph);
    if (parent[dest] == -1) {
      break;
    }

    int pathFlow = INT_MAX;
    for (int v = dest; v != src; v = parent[v]) {
      int u = parent[v];
      pathFlow = std::min(pathFlow, residualGraph[u][v]);
    }

    for (int v = dest; v != src; v = parent[v]) {
      int u = parent[v];
      residualGraph[u][v] -= pathFlow;
      residualGraph[v][u] += pathFlow;
    }

    maxFlow += pathFlow;
  }
```

```
  return maxFlow;
}
```

El algoritmo **Push-Relabel** se utilizó en el código para encontrar el flujo máximo entre dos colonias en un grafo ponderado debido a su eficiencia garantizada y su capacidad para manejar eficazmente redes de flujo grandes y complejas.

Implementación de Push-Relabel

```
// Estructura PushRelabel para implementar el algoritmo de flujo
máximo Push-Relabel
// basado en
https://www.geeksforgeeks.org/push-relabel-algorithm-set-2-implementa
tion/
struct PushRelabel {
  int V;
  std::vector<std::vector<int>> capacity;
  std::vector<std::vector<int>> flow;
  std::vector<int> height;
  std::vector<int> excess;
```

```
  PushRelabel(int V) : V(V), capacity(V, std::vector<int>(V, 0)),
flow(V, std::vector<int>(V, 0)), height(V, 0), excess(V, 0) {}

  void addEdge(int u, int v, int cap) {
    capacity[u][v] += cap;
  }

  void push(int u, int v) {
    int send = std::min(excess[u], capacity[u][v] - flow[u][v]);
    flow[u][v] += send;
    flow[v][u] -= send;
    excess[u] -= send;
    excess[v] += send;
  }

  void relabel(int u) {
    int minHeight = INT_MAX;
    for (int v = 0; v < V; ++v) {
      if (capacity[u][v] - flow[u][v] > 0) {
        minHeight = std::min(minHeight, height[v]);
        height[u] = minHeight + 1;
      }
    }
  }

  void discharge(int u) {
    while (excess[u] > 0) {
      bool pushed = false;
      for (int v = 0; v < V; ++v) {
        if (capacity[u][v] - flow[u][v] > 0 && height[u] == height[v]
+ 1) {
          push(u, v);
          pushed = true;
        }
      }
      if (!pushed) {
        relabel(u);
      }
    }
```

```
  }

  int getMaxFlow(int s, int t) {
    height[s] = V;
    excess[s] = 0;
    for (int v = 0; v < V; ++v) {
      if (capacity[s][v] > 0) {
        flow[s][v] = capacity[s][v];
        flow[v][s] = -flow[s][v];
        excess[v] += flow[s][v];
        excess[s] -= flow[s][v];
      }
    }

    std::queue<int> active;
    for (int i = 0; i < V; ++i) {
      if (i != s && i != t && excess[i] > 0) {
        active.push(i);
      }
    }

    while (!active.empty()) {
      int u = active.front();
      active.pop();
      discharge(u);
      if (excess[u] > 0) {
        active.push(u);
      }
    }

    return excess[t];
  }
};
```

Análisis de Complejidad / Justificación

El algoritmo de Edmonds-Karp es una implementación del método Ford-Fulkerson que sirve para encontrar el flujo máximo en una red de flujo. La red de flujo consiste en un grafo dirigido donde cada arista tiene una capacidad y dos nodos especiales, una fuente y un sumidero. El

objetivo del algoritmo es determinar la cantidad máxima de flujo que puede pasar de la fuente al sumidero sin exceder las capacidades de las aristas.
El algoritmo de Edmonds-Karp utiliza el método Ford-Fulkerson con búsqueda en anchura (BFS) para encontrar caminos aumentantes en la red de flujo. Un camino aumentante es un camino desde la fuente hasta el sumidero donde todas las aristas tienen capacidad residual positiva. La capacidad residual de una arista es la capacidad original menos el flujo actual. El algoritmo sigue encontrando y aumentando el flujo a lo largo de estos caminos hasta que no se pueden encontrar más caminos aumentantes.
Los pasos del algoritmo son los siguientes:

- Inicializar el flujo en todas las aristas a cero.
- Mientras haya un camino aumentante desde la fuente al sumidero (encontrado usando BFS):
- Determinar la capacidad residual mínima a lo largo del camino aumentante.
- Aumentar el flujo en ese valor a lo largo del camino.
- Actualizar las capacidades residuales de las aristas en el camino.
- Repetir el paso 2 hasta que no se puedan encontrar más caminos aumentantes.
- El flujo máximo es la suma del flujo en las aristas salientes de la fuente.

Complejidad temporal Edmonds-Karp

Primero se realiza el BFS, que como tiene que en el peor de los casos analizar todos los vertices y aristas, la complejidad es de **O(V+E)**. Después, cuando se implementa el algoritmos de Edmonds-Karp, se ejecuta otro bucle que itera, y este en el peor de los casos itera la misma cantidad de veces que el flujo de la red. Entonces en total, la complejidad temporal es de **O(VE^2)**

Por otro lado, el algoritmo de **Push-Relabel** es un método para encontrar el flujo máximo en una red de flujo. Este algoritmo se distingue de otros enfoques como el de Ford-Fulkerson en que no se basa en la búsqueda de caminos aumentantes desde la fuente hasta el sumidero. En su lugar, trabaja localmente en los vértices, empujando flujo entre ellos y ajustando sus alturas para permitir más movimientos de flujo.
El algoritmo de Push-Relabel se basa en dos operaciones fundamentales: "push" (empujar) y "relabel" (re etiquetar). En la fase de inicialización, el flujo se empuja desde la fuente hacia sus nodos adyacentes tanto como sea posible. Luego, el algoritmo procesa los vértices uno por uno, empujando el flujo de los vértices con exceso de flujo hacia sus vecinos más bajos en términos de altura. Si un vértice no puede empujar flujo debido a que todos sus vecinos tienen alturas mayores o iguales, se re etiqueta incrementando su altura. Este proceso se repite hasta que no haya más vértices con exceso de flujo.

Los pasos del algoritmo son los siguientes:

- Inicializar el flujo en todas las aristas a cero y la altura de todos los vértices a cero, excepto la fuente, cuya altura se establece en el número de vértices.
- Empujar el flujo desde la fuente hacia sus nodos adyacentes tanto como sea posible.
- Mientras haya vértices con exceso de flujo (nodos activos):
- Seleccionar un nodo activo.
- Intentar empujar flujo desde este nodo a sus vecinos si la capacidad residual lo permite y el vecino tiene menor altura.
- Si no se puede empujar flujo a ningún vecino, re etiquetar el nodo incrementando su altura.
- Repetir el paso 3 hasta que no haya más vértices con exceso de flujo.
- El flujo máximo es la suma del flujo en las aristas salientes de la fuente.

Complejidad temporal Push-Relabel

Las operaciones iniciales de capacity flow y height tienen complejidad de O(V^2). La operación de relabel recorre todos los vertices dando O(V). Y el push se realiza en un tiempo constante de O(1). Cada vez que se llama a la función de discharge se ejecuta un relabel y un push. Esto se hace hasta que el vertice sea 0. Entonces como el numero de veces que se llama es proporcional al numero de aristas E. Así que la complejidad temporal final es de **O(V^2E)**.

Parte 4

Descripción

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.

Implementación

El algoritmo de **Distancia Euclidiana** se utilizó en el código porque proporciona una medida precisa y fácilmente interpretable de la distancia entre la nueva central y las colonias existentes en un plano bidimensional. Es una medida directa y diagonal de la distancia entre los puntos.

El algoritmo de **Distancia de Manhattan** se utilizó en el código porque proporciona una medida eficiente y adecuada de la distancia entre la nueva central y las colonias existentes en un plano bidimensional. Es una medida de la distancia basada en movimientos horizontales y verticales a lo largo de una cuadrícula.

Implementación de Distancia Euclidiana

```
// Calcula la distancia mínima entre la central y las colonias usando
la
// distancia euclidiana
int distanciaMinimaEuclidiana(std::pair<int, int> src,
                              std::pair<int, int> dest) {
  return std::sqrt(std::pow(src.first - dest.first, 2) +
                   std::pow(src.second - dest.second, 2));
}
```

Implementación de Distancia de Manhattan

```
// Para comparar, vamos a calcular la distancia mínima entre la
central y las
// colonias usando la distancia de Manhattan
int distanciaMinimaManhattan(int src, int dest,
                             std::vector<std::pair<int, int>>
&coordinates) {
  return std::abs(coordinates[src].first - coordinates[dest].first) +
         std::abs(coordinates[src].second -
coordinates[dest].second);
}
```

Análisis de Complejidad / Justificación

La **Distancia Euclidiana** es una medida matemática que calcula la longitud de la línea recta entre dos puntos en un espacio euclidiano. Es decir, es la distancia "en línea recta" más corta entre dos puntos, representando intuitivamente la longitud del segmento de línea que los conecta. La distancia euclidiana se utiliza comúnmente en geometría, análisis de datos, y aprendizaje automático para medir la similitud o disimilitud entre dos puntos en un espacio multidimensional. En un espacio bidimensional (2D), si los puntos tienen coordenadas (x1, y1) y (x2, y2) la distancia euclidiana *d*d entre ellos se calcula usando la fórmula:

$$d = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

Los pasos del algoritmo son los siguientes:

- Definir los Puntos - Cada punto en el plano se representa por sus coordenadas (x, y). Supongamos que tenemos dos puntos, P1 y P2.
- Restar las coordenadas correspondientes - Calcular la diferencia entre las coordenadas x y y de los dos puntos.
- Elevar al Cuadrado las Diferencias
- Sumar los Cuadrados de las Diferencias
- Calcular la Raíz Cuadrada de la Suma

Complejidad Temporal de Distancia Euclidiana

Aunque dentro de las operaciones para obtener la distancia incluye dos cuadrados y una raíz, la complejidad es **O(n)** porque esas operaciones se hacen solo una vez y no se iteran.

Por otro lado, la **Distancia de Manhattan**, también conocida como distancia de bloques o distancia *L1*, es una medida matemática que calcula la distancia entre dos puntos a lo largo de los ejes del sistema de coordenadas. Se llama así porque evoca la manera en que uno debe recorrer las calles de una cuadrícula de ciudad, como en el distrito de Manhattan en Nueva York, donde uno debe seguir calles perpendiculares en lugar de moverse en línea recta.
La distancia de Manhattan se utiliza comúnmente en geometría, análisis de datos y aprendizaje automático para medir la diferencia entre puntos en un espacio con restricciones de movimiento ortogonales. En un espacio bidimensional (2D), si los puntos tienen coordenadas ($x1$,$y1$) y ($x2$,$y2$), la distancia de Manhattan dd entre ellos se calcula usando la fórmula:

$$d = |x2 - x1| + |y2 - y1|$$

Los pasos del algoritmo son los siguientes:

- Definir los Puntos - Cada punto en el plano se representa por sus coordenadas (x, y). Supongamos que tenemos dos puntos, P1 y P2.
- Calcular la Diferencia Absoluta de las Coordenadas
- Sumar las Diferencias Absolutas

Complejidad Temporal de Distancia de Manhattan

Como solo hay suma y resta absoluta de los puntos, entonces la complejidad temporal es de **O(n)**.

Reflexión

Eliuth Balderas Neri

Cómo se pudo observar en la presente actividad. Hay diferentes algoritmos que nos permiten resolver los problemas anteriormente descritos. No obstante, es de suma importancia tomar el contexto completo de cada uno de los problemas, para que de esta manera se pueda elegir el mejor algoritmo según las necesidades del problema. Esto debido a que en algunos casos, hay algoritmos que tienen una complejidad computacional menor y por ende, le toma menos tiempo a la computadora compilar. Sin embargo, pueden existir algoritmos menos rápido pero con una solución muchísimo más óptima. Adicionalmente, hay algoritmos que funcionan mejor en casos hipotéticos que en la vida real, por lo que es necesario saber cómo es que se van a aplicar estos algoritmos. Así pues, el tiempo de compilación o complejidad computacional no siempre son el factor más importante a considerar, sino que realmente se debe considerar todo el contexto.

Luis Gabriel Delfín Paulín

Como ya visto en el proyecto, el uso de diferentes algoritmos fue una parte fundamental para el análisis de los resultados a la situación problema. Cada uno nos proporciona un nivel de optimización diferente y esto nos permitió abordar los problemas desde diferentes perspectivas, pudiendo tener soluciones más eficaces y adaptadas a las necesidades de nuestra problemática. Esto no significa que los algoritmos que no se usaron, no sean útiles, es importante destacar que depende del contexto el tipo de algoritmo que se usó. Por ejemplo Kruskal y Prim que son algoritmos de Árbol de Expansión Mínima (MST), su eficiencia varía dependiendo de la estructura del grafo. Prim es más eficiente para grafos densos mientras que Kruskal es más eficiente para grafos dispersos.

Jorge Emiliano Pomar Mendoza

Con el objetivo de dar solución a la problemática presentada, se llevó a cabo una investigación de los algoritmos acerca de buscar cuales algoritmos vistos en clase podían satisfacer la necesidad de llevar la red de telecomunicaciones entre varias ciudades. Nuestro principal enfoque en este caso fue el tiempo. Así que investigamos, y tomamos la decisión de elegir estos algoritmos para cada una de los problemas de la actividad. Al final, nos dimos cuenta que en este caso fue muy pequeña la diferencia de tiempo entre un algoritmo y otro. En algunos caso como en el de las distancias se podría decir que hasta no era totalmente fiel la representación de la velocidad por varios factores de compilación y de las operaciones. Sin embargo, al haber reflexionado un poco más, creo que si bien el tiempo en este caso no fue nuestro indicador más fiel, definitivamente me ayudó a comprender la funcionalidad de los algoritmos al menos a un nivel de indicador. En el que podía ver el comportamiento de las optimizaciones de cada algoritmo a través del tiempo.

Referencias

GeeksforGeeks. (n.d.). Branch and Bound Algorithm. GeeksforGeeks. Recuperado de https://www.geeksforgeeks.org/branch-and-bound-algorithm/

GeeksforGeeks. (n.d.). Prim’s Minimum Spanning Tree (MST) | Greedy Algo-5. GeeksforGeeks. Recuperado de https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

GeeksforGeeks. (n.d.). Push Relabel Algorithm | Set 2 (Implementation). GeeksforGeeks. Recuperado de https://www.geeksforgeeks.org/push-relabel-algorithm-set-2-implementation/?ref=header_search

GeeksforGeeks. (n.d.). Kruskal’s Minimum Spanning Tree Algorithm | Greedy Algo-2. GeeksforGeeks. Recuperado de https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/?ref=header_search

GeeksforGeeks. (2022, 23 junio). Difference between Prim s and Kruskal s algorithm for MST. GeeksforGeeks. https://www.geeksforgeeks.org/difference-between-prims-and-kruskals-algorithm-for-mst/?ref=lbp

Shiksha. (n.d.). Key Differences Between Prim’s and Kruskal’s Algorithm. Shiksha. Recuperado de https://www.shiksha.com/online-courses/articles/difference-between-prims-and-kruskal-algorithm-blogId-155863#:~:text=Key%20Differences%20Between%20Prims%20and%20Kruskal%20Algorithm,-Here%20are%20the&text=Kruskal's%20algorithm%20can%20work%20with,for%20managing%20vertices%20and%20edges.