

JUGADOR 1



PUNTUACIÓN MÁS ALTA 2500

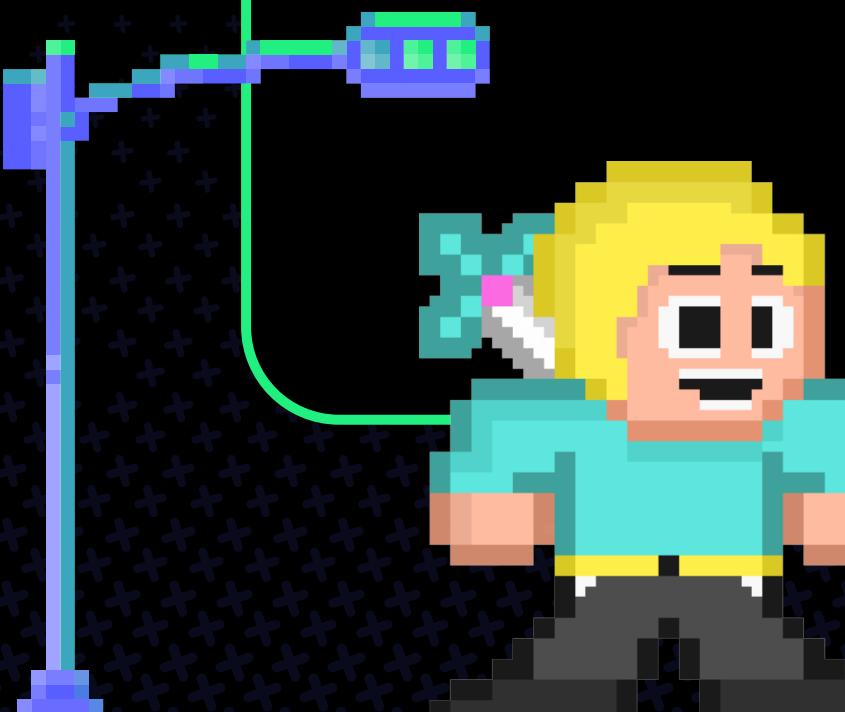


JUGADOR 2

# ACTIVIDAD INTEGRADORA

2

START



- ◆ MARÍA FERNANDA MORENO GOMÉZ
- ◆ JOSE RICARDO ROSALES CASTAÑEDA
- ◆ URI JARED GOPAR MORALES



MENU

➡ 01

♦ 07

★ 12



# CONTENIDO



DESCRIPCION



ETAPA 1  
DIJKSTRA



ETAPA 2  
NEAREST NEIGHBOR



ETAPA 3  
FORD FULKERSON



ETAPA 4  
DISTANCIA  
EUCLIDIANA



# DESCRIPCIÓN



◆ 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.

# DÍJKSTRA

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

```

30 //           vector<bool>& inMST: vector with the vertices included in MST.
31 // Return value: int minIndex: index of the vertex with the minimum distance.
32 // Complexity: O(n)
33 // -----
34 int minDistance(vector<int>& dist, vector<bool>& inMST) {
35     int minDist = INT_MAX, minIndex;
36
37     for (int i = 0; i < dist.size(); i++) {
38         if (!inMST[i] && dist[i] < minDist) {
39             minDist = dist[i];
40             minIndex = i;
41         }
42     }
43
44     return minIndex;
45 }
46
47 // -----
48 // Function: dijkstra
49 // Description: This function finds the shortest paths from a source vertex to
50 //               all other vertices in the graph.
51 // Parameters: vector<vector<int>>& graph: adjacency matrix of the graph.
52 //               int src: source vertex.
53 // Return value: vector<int> dist: vector with the distances from the source.
54 // Complexity: O(n^2)
55 // -----
56 vector<int> dijkstra(vector<vector<int>>& graph, int src) {
57     int numColonies = graph.size();
58     vector<int> dist(numColonies, INT_MAX);
59     vector<bool> inMST(numColonies, false);
60
61     dist[src] = 0;
62
63     for (int count = 0; count < numColonies - 1; count++) {
64         int u = minDistance(dist, inMST);
65         inMST[u] = true;
66
67         for (int v = 0; v < numColonies; v++) {
68             if (!inMST[v] && graph[u][v] && dist[u] != INT_MAX
69                 && dist[u] + graph[u][v] < dist[v]) {
70                 dist[v] = dist[u] + graph[u][v];
71             }
72         }
73     }
74
75     return dist;
76 }
77

```

Para esta solución vamos a ocupar el algoritmo de Dijkstra por que nos ayuda a encontrar las rutas más cortas desde un nodo inicial al los demás nodos, gracias a esto minimizaremos la longitud del cable.

Este algoritmo funciona iniciando un nodo fuente con el valor de 0 y todos los demás tendran una distancia infinita solo al inicio.

Luego se examinan cada arista saliente del nodo actual. Y si la distancia al nodo al final de la arista es mayor a la suma de la distancia del nodo actual y el peso de la arista, entonces se actualiza la distancia al nodo final.

Ahora que tenemos todas las aristas del nodo actual examinadas simplemente se selecciona la no visitada por la distancia más corta el cual sera uno de los vecinos directos de nuestro nodo fuente.

Todo este proceso se repite hasta que al final tendremos las distancias más cortas desde el nodo fuente a los demás nodos

Complejidad:  $O(n^2)$ ,

# NEAREST NEIGHBOR

¿cuál es la ruta más corta posible que visita cada colonia exactamente una vez y al finalizar regresa a la colonia origen?

Decidimos ocupar nearest neighbor porque proporciona los resultados rápidos, lo cual es beneficioso si tenemos un número de colonias grande.

Su funcionamiento es que al iniciar con un nodo lo añade a la ruta, entre cada paso selecciona el vecino más cercano y lo añade a la ruta , repite el proceso hasta que todas las colonias sean visitadas.

Como final añade la colonia de origen al final del recorrido para completar el ciclo.

Calcula el costo total al recorrer el recorrido suma las distancias entre colonias consecutivas y imprime la distancia total del recorrido

complejidad de  $O(n^2)$

```
// Parameters: vector<vector<int>>& graph: adjacency matrix of the graph.  
// Return value: vector<int> tour: vector with the order of the vertices.  
// Complexity: O(n^2)  
// ======  
vector<int> nearestNeighbor(const vector<vector<int>>& graph) {  
    int n = graph.size();  
    vector<int> tour;  
    vector<bool> visited(n, false);  
  
    int current = 0;  
    tour.push_back(current);  
    visited[current] = true;  
  
    for (int i = 1; i < n; ++i) {  
        int next = -1;  
        double minDistance = numeric_limits<double>::infinity();  
  
        for (int j = 0; j < n; ++j) {  
            if (!visited[j]) {  
                double distance = graph[current][j];  
                if (distance < minDistance) {  
                    minDistance = distance;  
                    next = j;  
                }  
            }  
        }  
  
        tour.push_back(next);  
        visited[next] = true;  
        current = next;  
    }  
  
    tour.push_back(tour[0]);  
  
    return tour;  
}
```

# FORD FULKERSON

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

```
// Function: FordFulkerson
// Description: This function finds the maximum flow from source 's' to sink
//              't' in the graph.
// Parameters: vector<vector<int>>& graph: adjacency matrix of the graph.
//              int s: source vertex.
//              int t: sink vertex.
// Return value: int: maximum flow from source 's' to sink 't' in the graph.
// Complexity: O(n^2)
// -----
int fordFulkerson(vector<vector<int>>& graph, int s, int t) {
    int numColonies = graph.size();
    vector<vector<int>> rGraph(numColonies, vector<int>(numColonies));

    for (int u = 0; u < numColonies; u++) {
        for (int v = 0; v < numColonies; v++) {
            rGraph[u][v] = graph[u][v];
        }
    }

    vector<int> parent(numColonies);
    int maxFlow = 0;

    while (bfs(rGraph, s, t, parent)) {
        int pathFlow = INT_MAX;

        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            pathFlow = min(pathFlow, rGraph[u][v]);
        }

        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            rGraph[u][v] -= pathFlow;
            rGraph[v][u] += pathFlow;
        }

        maxFlow += pathFlow;
    }

    return maxFlow;
}

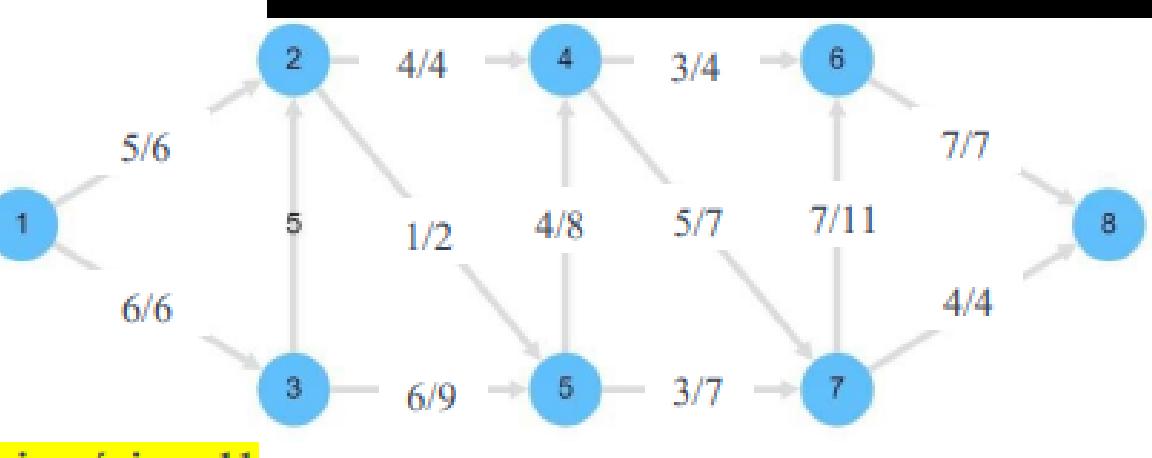
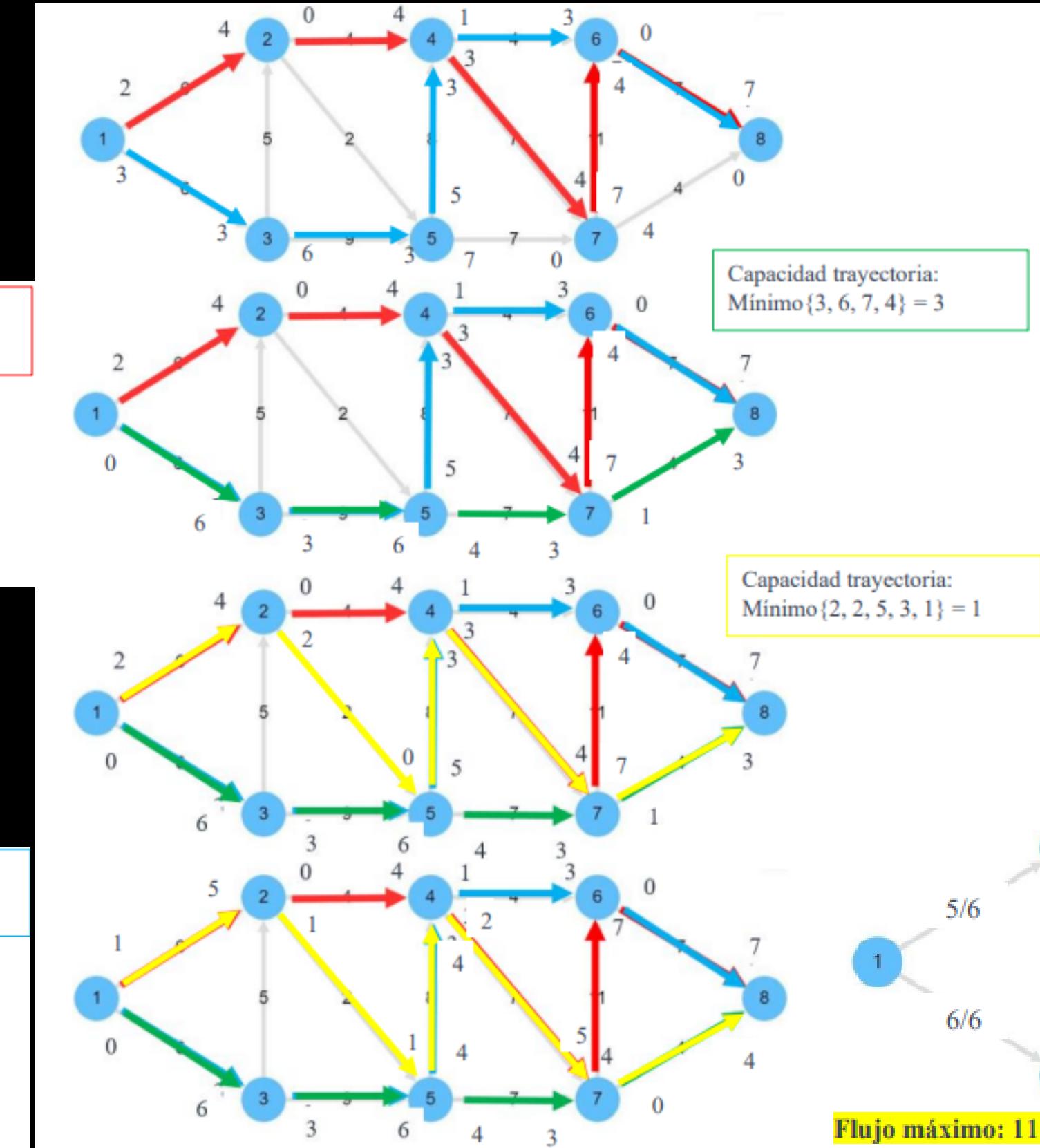
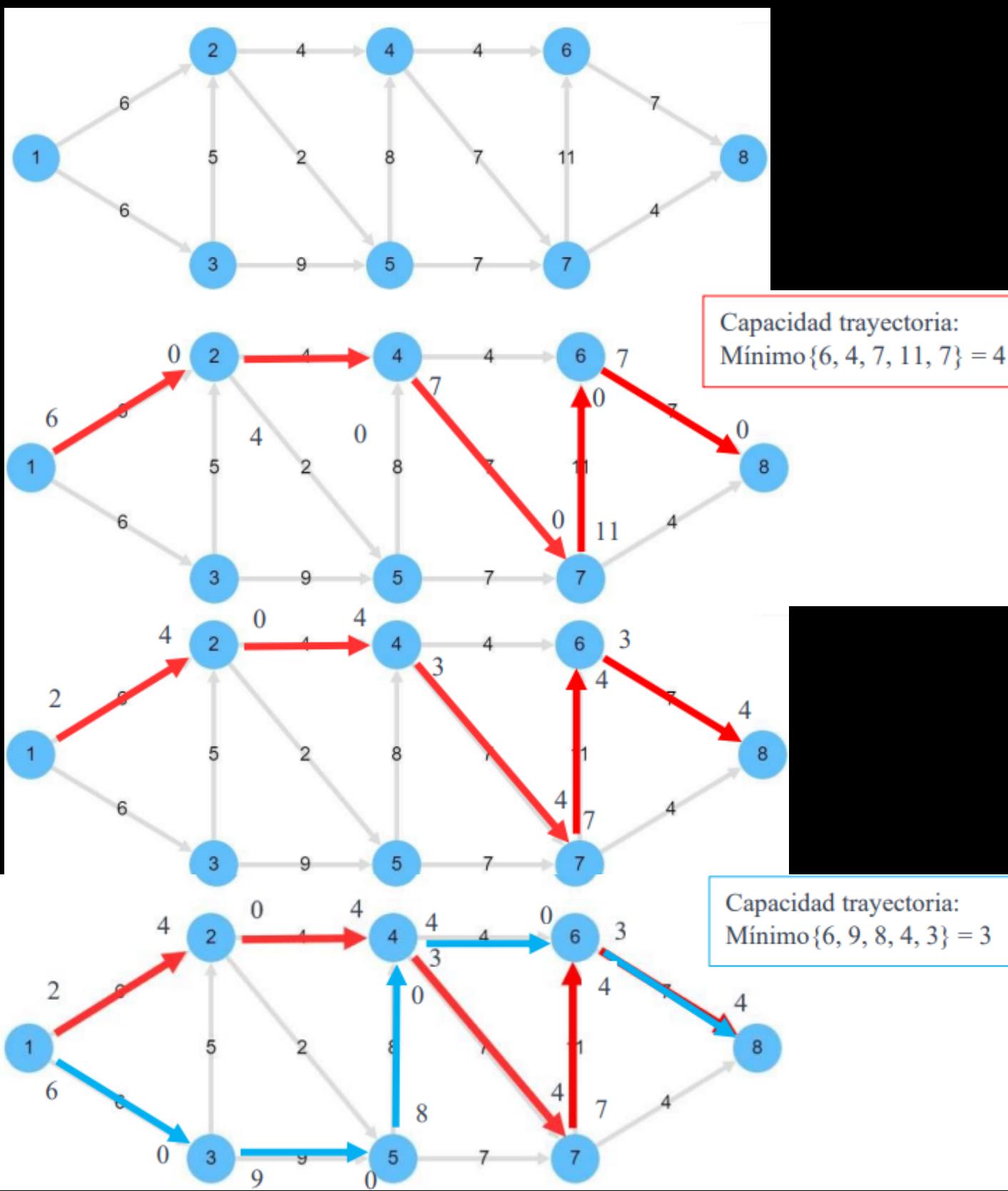
// -----
// Function: print max flow
// Description: This function prints the maximum flow from source 's' to sink 't'.
// Parameters: vector<vector<int>>& graph: adjacency matrix of the graph.
//              int s: source vertex.
//              int t: sink vertex.
```

Este algoritmo es adecuado para calcular el flujo máximo de transmisión de datos entre colonias, también maneja redes donde las capacidades de los enlaces varían como por ejemplo por las interferencias electromagnéticas y encuentra la solución más óptima.

Inicializamos un vector para llevar el registro de los vértices visitados. Luego utilizamos la búsqueda en anchura BFS desde el vértice fuente mientras haya un camino aumentativo de 's' a 't'. Encontramos el flujo mínimo en este camino, actualizamos las capacidades residuales a lo largo de este camino.

Por lo que si llega al vértice sumidero 't' retorna 'true' indicando el camino  
La complejidad es de  $O(n^2)$ , donde 'n' es el número de vértices (colonias) en el grafo.

# FORD FULKERSON



# DISTANCIA EUCLIDIANA

*¿Podríamos analizar la factibilidad de conectar a la red un nuevo punto (una nueva localidad) en el mapa ?ctamente una vez y al finalizar regresa a la colonia origen?*

La distancia euclidiana nos ayuda a medir la distancia en línea recta entre dos puntos de un espacio, por lo que ofrece una estimación inicial sin que varie debido a la topografía.

Para implementarla necesitamos ocupar la fórmula de la distancia entre dos puntos  $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Y el funcionamiento del código es que inicializamos el índice de punto más cercano y la distancia mínima, luego recorre todos los puntos en el vector, calculando la distancia de cada uno al punto de referencia.

Por último actualiza el punto más cercano y la distancia mínima y retorna el índice del punto más cercano

**Complejidad:**  $O(n)$

```

35 // -----
36 double calculateDistance(int x1, int y1, int x2, int y2) {
37     return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));
38 }
39
40 // -----
41 // Function: euclideanDistance
42 // Description: This function finds closest point in a vector to
43 //                 a given point.
44 // Parameters: vector<pair<int, int>>& points: vector with the points.
45 //                 int x: x coordinate of the point.
46 //                 int y: y coordinate of the point.
47 // Return value: int closestPoint: index of the closest point.
48 // Complexity: O(n)
49 // -----
50 int euclideanDistance(vector<pair<int, int>>& points, int x, int y) {
51     int closestPoint = 0;
52     double minDistance = numeric_limits<double>::infinity();
53
54     for (int i = 0; i < points.size(); i++) {
55         double distance = calculateDistance(x, y, points[i].first, points[i].second);
56         if (distance < minDistance) {
57             minDistance = distance;
58             closestPoint = i;
59         }
56     }
57
58     return closestPoint;
59 }
50
56 // -----
57 // Function: printClosest
58 // Description: This function prints the closest point to a given point.
59 // Parameters: vector<pair<int, int>>& points: vector with the points.
60 //                 int x: x coordinate of the point.
61 //                 int y: y coordinate of the point.
62 // Return value: void.
63 // Complexity: O(n)
64 // -----
65 void printClosest(vector<pair<int, int>>& points, int x, int y) {
66     int closestPoint = euclideanDistance(points, x, y);
67     cout << "La central mas cercana a [" << x << ", " << y << "] es [" <<
68     points[closestPoint].first << ", " << points[closestPoint].second << "] " <<
69     "con una distancia de " << calculateDistance(x, y, points[closestPoint].first,
70     points[closestPoint].second) << "." << endl;
71 }
72
73 #endif // ECULIDEANDISTANCE_H

```

MENU



MUCHAS  
GRACIAS!