Actividad Integradora II

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

Agenda

01 Problemática

02 Parte 1

03 Parte 2

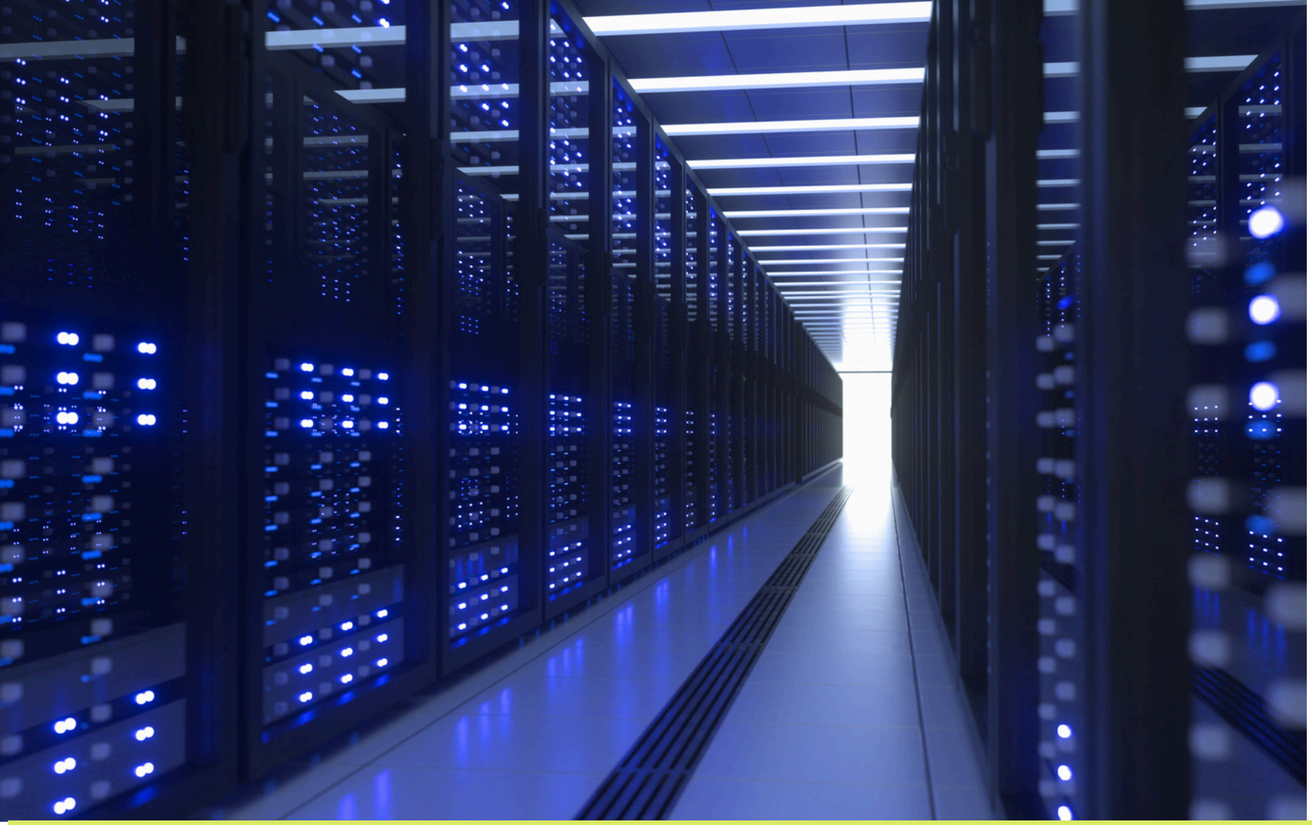
04 Parte 3

05 Parte 4

06 Resultados/Justificación

07 Conclusión

08 Referencias



En México existe una falta de infraestructura tecnológica que provoca una desigualdad en cuanto al acceso a internet.

Problemática

Parte 1

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.

Kruskal

VS

Prim

Kruskal $O(ElogE)+O(E\alpha(V))$

- 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.

```cpp
// 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;
}
```

```
==========================
====== Parte 1 ======
==========================
Forma óptima de cablear las colonias:
C - D (7 km)
A - B (16 km)
B - C (18 km)
Tiempo de ejecución (Kruskal): 0.00275 ms
```

Prim

O((V+E) log V)

El algoritmo incrementa gradualmente el tamaño del árbol.

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.

```cpp
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);
}
```

```
==========================
====== Parte 1 ======
==========================
Forma óptima de cablear las colonias:
A - B (16 km)
B - C (18 km)
C - D (7 km)
Tiempo de ejecución (Prim): 0.06795 ms
--------------------------
```

Parte 2

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.

TSP - Nearest Neighbor

VS

Branch and Bound Algorithm

TSP - Nearest Neighbor O(N^2)

Este algoritmo 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, se mueve 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.

```cpp
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;
}
```

```
======================
====== Parte 2 ======
======================
Ruta del Viajero:
A B C D A
Distancia total: 82 km
Tiempo de ejecución (TSP - Nearest Neighbor): 0.01482 ms
```

Branch and Bound Algorithm O(N!) O(N^3)

Los pasos del algoritmo pueden describirse como sigue:

1. Inicializar la mejor solución como infinita y crear un nodo raíz con la ciudad inicial y una cota inferior calculada.
2. Insertar el nodo raíz en una cola de prioridad, ordenada por cota inferior.
3. 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.
4. La mejor solución encontrada al finalizar el proceso es la ruta óptima.

```cpp
// 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);
            }
```

```
======================
====== Parte 2 ======
======================
Ruta del Viajero:
A B C D A
Distancia total: 73 km
Tiempo de ejecución (TSP - Branch and Bound): 0.0175 ms
```

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.

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

Edmonds-Karp

VS

Push-Relabel

Edmonds-Karp O(VE^2)

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.

```cpp
// Primero se hace un BFS para encontrar un camino que va en aumento y con
// capacidad residual mayor a 0
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;
}
```

```cpp
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;
  }
```

```
========================
====== Parte 3 ======
========================
Flujo máximo entre las colonias 0 y 3: 78 personas
Tiempo de ejecución (Edmonds-Karp): 0.0022 ms
------------------------
```

Push-Relabel O(V^2E)

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.

```
  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;
        }
```

```
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];
}
;
```

```
======================
====== Parte 3 ======
======================
Flujo máximo entre las colonias 0 y 3: 78 personas
Tiempo de ejecución (Push-Relabel): 0.00057 ms
----------------------
```

Parte 4

Cuál es la central más cercana geográficamente a una nueva contratación. No necesariamente hay una central por cada colonia. Se pueden tener colonias sin central, y colonias con más de una central.

Distancia Euclidiana

VS

Manhattan

Distancia Euclidiana O(n)

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

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

```cpp
// Calcula la distancia minima 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));
}
```

Manhattan O(n)

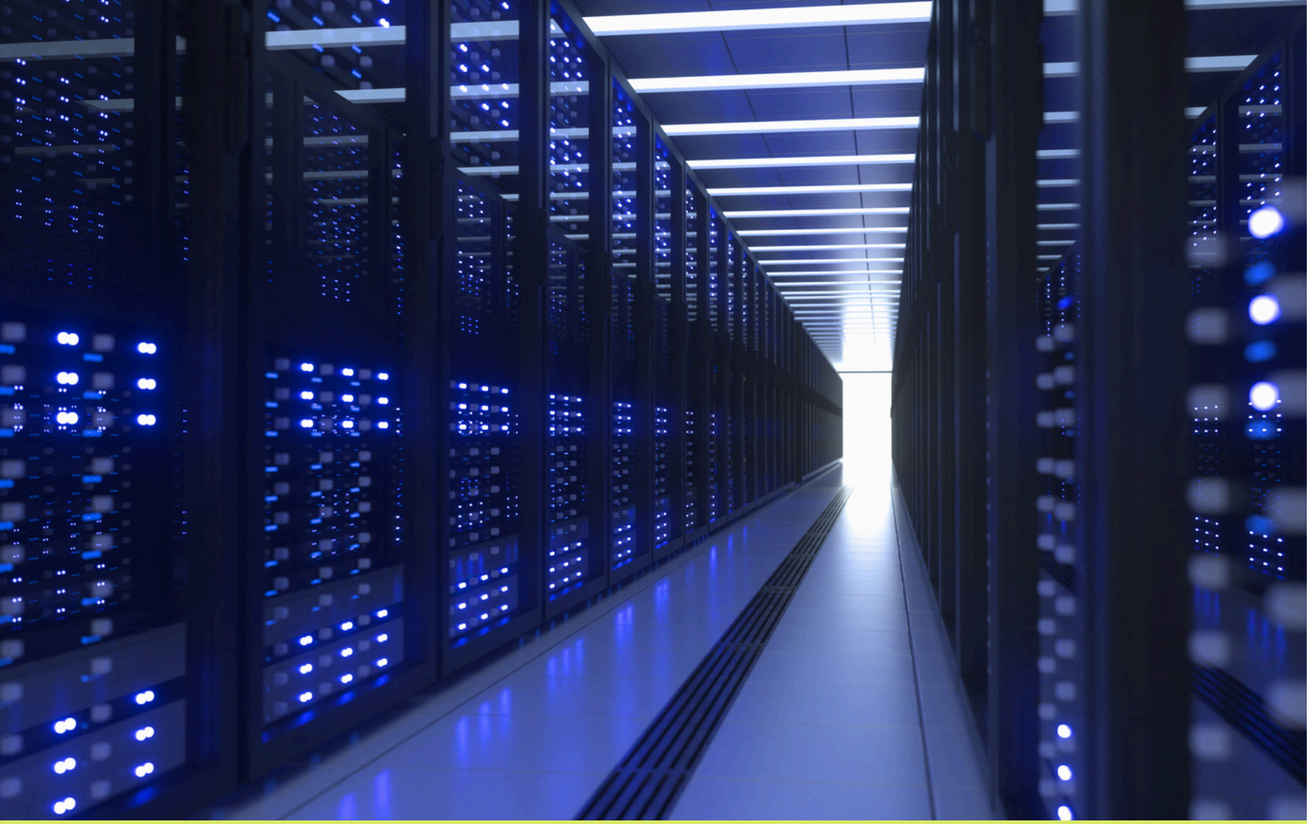
Los pasos del algoritmo son los siguientes:

- 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

$$d = |x_2 - x_1| + |y_2 - y_1|$$

```cpp
// Para comparar, vamos a calcular la distancia minima 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);
}
```

```
======================
====== Parte 4 ======
======================
Distancia mínima entre la nueva central y las colonias existentes (euclidiana):
A: 825638968 km
B: 320 km
C: 439 km
D: 337 km
Tiempo de ejecución (Distancia Euclidiana): 0.000169 ms
Distancia mínima entre la nueva central y las colonias existentes (Manhattan):
A: 825639320 km
B: 700 km
C: 400 km
D: 600 km
Tiempo de ejecución (Distancia Manhattan): 0.00018 ms
```



Conclusión

Se presentaron distintas implementaciones de algoritmos diferentes de optimización en un sistema. Cada algoritmo cumple un propósito específico y ofrece soluciones eficientes para problemas de conectividad, ruta y capacidad de flujo.

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.