Problema K-minimum spanning tree

Roberto Juan Cayro Cuadros, Gabriel Alexander Valdivia Medina, Giulia Alexa Naval Fernández, Rodrigo Alonso Torres Sotomayor

Universidad Católica San Pablo

Resumen

El presente trabajo presenta una breve investigación del problema *k-minimun spanning tree*, explicando su funcionamiento, demostrando que pertenece al conjunto de los problemas NP-completos, y dando opciones de algoritmos para su resolución.

1. Conocimientos previos

1.1. Problemas de la clase P

Los problemas de la clase P (Polynomial time) son todos aquellos que se pueden resolver en tiempo polinomial. Es decir, pueden ser resueltos polinomialmente en el mundo real. Entre los problemas más conocidos se encuentran la búsqueda del elemento mínimo, la ordenación de un conjunto de elementos, encontrar un árbol mínimo de expansión, etc.

1.2. Problemas de la clase NP

Los problemas de la clase NP (Non-deterministic polynomial time) son aquellos que pueden ser resueltos en tiempo polinomial usando una máquina o un algoritmo **no determinístico**. En la mayoría de casos, estos algoritmos no se pueden representar adecuadamente en la vida real por su carácter no determinístico. Sin embargo, los problemas NP se pueden verificar con facilidad, siendo verificables en tiempo polinomial. Algunos ejemplos conocidos pueden ser el camino Hamiltoniano, la coloración de grafos, entre otros.

1.3. Relación entre NP y P

Uno de los problemas más famosos de la computación es el determinar si $P = NP$. Se sabe que $P \subset NP$, ya que ambos pueden comprobarse en tiempo polinomilal. Sin embargo, para demostrar que $P = NP$ se tendría que demostrar que existe una solución polinomial para los NP, cosa que no ha sido demostrada hasta la fecha y que se cree que nunca lo será.

1.4. Problemas de la clase NP-hard

A pesar del nombre, no todos los problemas NP-hard pertenecen a NP. La principal característica de estos problemas es que son por lo menos tan difíciles como el problema NP más difícil. Además, todo problema que pertenece a la clase NP se puede transformar o reducir a un problema NP-hard. Por otro lado, estos problemas son mucho más difíciles de verificar que los NP. Algunos ejemplos conocidos son el problema de detención (halting problem) y hallar un camino **no Hamiltoniano**.

1.5. Problemas de clase NP-completo

Los NP- completo son un tipo especial de problema, todos los problemas NP-completo pertenecen a su vez tanto a los NP como a los NP-hard. Su peculiaridad principal es que todo problema NP-completo puede reducirse a cualquier otro problema NP-completo en **tiempo polinomial**. Por lo tanto, si se pudiera encontrar una solución polinomial para cualquier problema NP-completo, entonces se podrían encontrar también soluciones polinomiales para todos los problemas del conjunto. Algunos problemas conocidos son el ciclo Hamiltoniano, SAT, entre otros. La manera más fácil de demostrar que un problema pertenece a los NP-completos es primero demostrar que pertenece a NP, con un algoritmo no determinístico en tiempo polinomial. Y luego, demostrar que un problema NP-completo ya conocido puede transformarse para ser resuelto por el algoritmo de este problema.

2. Introducción al problema k-MST

Según múltiples fuentes[3][5], el k-MST o k-minnimum spanning tree problem, árbol de expansión de péso mínimo k en español es un problema computacional que pide un árbol de mínimo costo con exactamente k vértices que forme un subgrafo del grafo original.

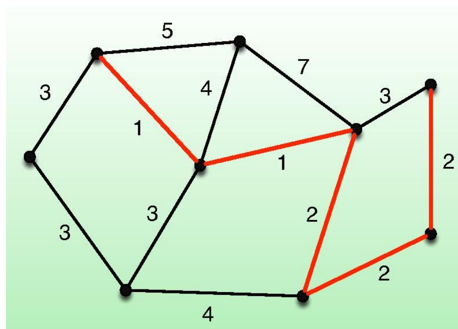

Figura 1: 6-MST del grafo G. Fuente: Wikipedia Commons

3. Demostración NP-completo

No es posible suponer la naturaleza del problema, y establecer que es NP-hard o NP-completo, sin la evidencia correspondiente, para probrarlo este debe pertenecer a NP, ademas que un problema NP-completo pueda reducirse al mismo.

3.1. Demostrar que k-MST $\in$ NP

Para demostrar que un problema pertenece a la clase NP, se debe crear un algoritmo no deterministico que resuelva el problema en tiempo polinomial:

k-MST (G,k)

```
t ← 0
while t < k
      do u ←ESCOGER(G)
            if u is not in x
                  do x . add(u)
                   t++
```

X será el árbol a construirse,junto con el bucle while y una variable t confirmaremos la adición de exactamente k vértices, escogeremos algún u de G, si u ya esta dentro de x el valor de t no cambie por lo tanto se repite hasta tomar otro vértice.

3.2. Transformación NP-completo α k-MST

El segundo paso para demostrar que un problema pertenece a los NP-completos, es transformar un problema NP-completo conocido para que pueda ser resuelto por el algoritmo del k-MST. Una transformación sencilla es la que se puede hacer dese el problema de Steiner.

3.2.1. Steiner problem

Según el artículo de Shivam Gupta[2], el Steiner problem es un problema NP-completo de los 21 problemas de Karp, usado en problemas de optimización y mayormente enfocado en estructuras de grafos aunque tambien visto en aplicaciones de modelación de redes con más de 2 terminales. El problema consiste en que, dado un grafo no-dirigido de aristas con peso, generar un arbol dado un *Sub-set* de vertices los cuales formarán este arbol. Además, pueden añadirse nuevos vertices del grafo al *sub-set* para lograr las conexiones entre estos, llamados *Steiner-vertices.*

La decisión asociada al problema será averiguar si existe un árbol que una todos los vértices de un *sub-set* R, usando máximo M aristas. Los vertices deberán ser exactamente los dados en el Sub-set. Esta decisión es conocida por ser del grupo de los NP-completos. La principal diferencia con el k-MST es que aquí recibimos un conjunto específico de vectores para conformar nuestro árbol, pudiendo usar vértices fuera de la relación para conectarlos. El k-MST no recibe esta relación, sólo el número de vértices exactos que necesita.

3.3. Entradas y salidas

Steiner-tree

Steiner-problem
Entrada: *Grafo no-dirigido G con aristas de peso. *Sub-set de vertices R. *Número M.
Salida: *Arbol de menor peso con los vertices de S y los Steiner-vertices si fueran necesarios.

k-MST

k-MST
Entrada: *Grafo no-dirigido G con aristas de peso. *Número k de vértices.
Salida: *Arbol de menor peso con k-vertices y k-1, aristas.

3.3.1. Transformación

Una primera aproximación será que dada la entrada G para Steiner, se puede tomar el mismo grafo para k-MST, puesto que tiene las aristas pesadas y un número determinado de vertices. De esta forma aseguramos la transformación y no afectara la salida porque siempre busaremos el arbol de menor peso , se usará el tamaño del sub-set de vertices siendo este igual a k.

Pero no podemos asegurar que esta transformación pudiera también resolver al Steiner tree, siendo esta una de las propiedades en una transformación polinómica. Como tenemos de entrada un G, y k, podriamos calcular todas las permutaciones de G en k. Y necesariamente una de ellas corresponderia a la solución para Steiner:

$$\text{Total de permutaciones (Tn)} = \binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

$$S \subseteq Tn.$$

Sin embargo, calcular todas las permutaciones de una cantidad n de elementos es un proceso con una complejidad $O(!n)$, que no entra dentro de complejidad polinomial, además que esta reducción planteada no resolverá el problema de k-MST, ya que este necesitaría solo 1 árbol de menor peso. Es necesario entonces otro tratamiento para que el k-MST opere con los vértices que el algoritmo Steiner pide. Siguiendo la transformación de R. Ravi [4], otra idea es añadir un árbol con aristas de peso 0 en cada vértice que pertenezca a R, y transformar k como $k = |R|(X+1)$, siendo X la cantidad de vértices que tendrán cada uno de estos árboles, denotado como $X = |V(G)| - |R|$ De esta forma, el k-MST utilizará los vértices de R sí o sí como parte de su solución.

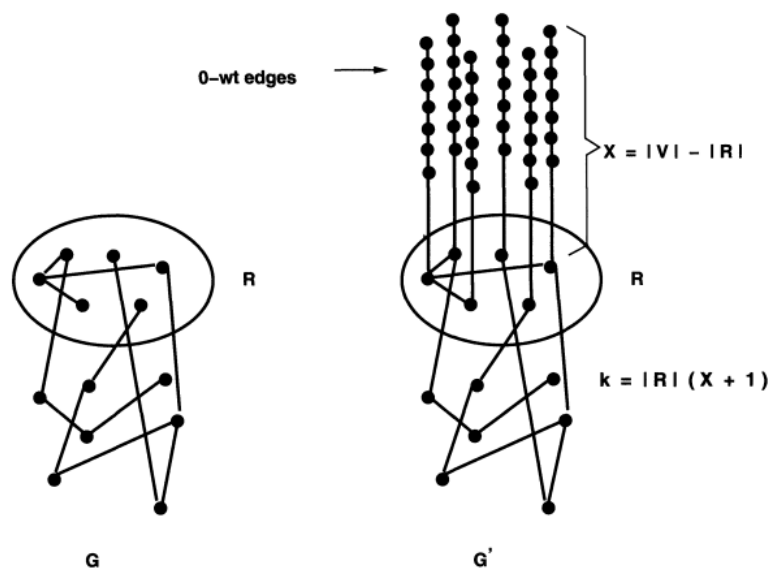

Figura 2: Transformación de la entrada de Steiner a entrda de k-MST. Fuente: www.contrib.andrew.cmu.edu

En este nuevo grafo G', las aristas que unen los nuevos vértices de X tendrán un peso de 0, las aristas correspondientes a las aristas originales de G tendrán un peso de 1, y el resto de pares del grafo tendrán un peso de ∞. De este modo, el algoritmo del k-MST encontrará el árbol de menor peso con el parámetro k en G', y verificará si es de igual o menor peso que M, satisfaciendo el requisito de las M aristas debido a que estas tendrán peso 1.

Por otra parte cumpliremos la propiedad de la transformación polinomial, en donde redujimos el problema de Steiner a k-MST, por ello la solución a k-MST mediante otra transformación polinómica podra ser la solución a Steiner-tree, gracias al valor de la transformación de k= a (X+1)—R—, puesto que las aristas de los árboles agregados son de peso 0, estos serán agregados a la solución y por el k se confirma que en la solución de k-MST estarán los vértices de R.

4. Algoritmo de fuerza bruta

Para el algoritmo de fuerza bruta, es suficiente una modificación al algoritmo Prim convencional, limitando su avance a k nodos y haciendo que se repita con cada nodo del grafo G como origen. Finalmente, decidir qué arbol de todos los obtenidos ha sido el más corto.

4.1. Algoritmo de prim.

Es un *algoritmo greedy*, que dado un grafo G encuentra el MST(Minimun-spanning-tree) de menor peso posible, usando todos los vértices de G y donde el peso total es el mínimo. El algoritmo funciona un vértice a la vez buscando la conexión al siguiente vértice de menor peso de la forma:

1. Iniciar el árbol con un vértice cualquiera del grafo
2. Construir el árbol, vértice por vértice usando aquellos vértices que ya no estén agregados
3. repetir el paso 2 hasta que todos los vértices estén en el árbol

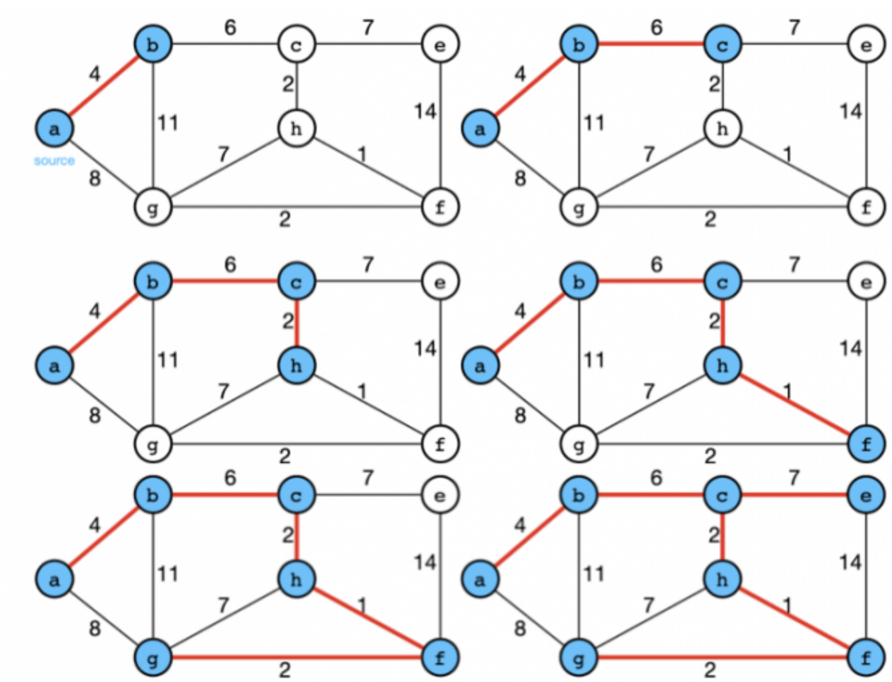

Figura 3: Funcionamiento de prim. Fuente: https://laptrinhx.com/minimum-spanning-tree-prim-4207877151/

4.2. Implementación

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
typedef vector<int> valNode;
typedef vector<valNode> adyacencias;

int PrimsMST(int sourceNode, vector<adyacencias>& graph, int K)
{
    //Guardar detalles del nodo.
    priority_queue<valNode, vector<valNode>, greater<valNode>> k;
    int count = 0;
    vector<int> aux = { 0,sourceNode };
    k.push(aux);
    bool* nodesAdded = new bool[graph.size()];
    memset(nodesAdded, false, sizeof(bool) * graph.size());
    int mst_tree_cost = 0;

    while (count != K)
    {
        // Nodo con  m nimo  costo
        valNode itemNode;
        itemNode = k.top();
        k.pop();
        int Node = itemNode[1];
        int Cost = itemNode[0];

        //Checar si el nodo ya se   a adi
        if (!nodesAdded[Node])
        {
            mst_tree_cost += Cost;
            count++;
            if (count == K)
                break;
            nodesAdded[Node] = true;

            //Nodos vecinos quitados de priority queque
            for (auto& node_cost : graph[Node])
            {
                int adjacency_node = node_cost[1];
                if (nodesAdded[adjacency_node] == false)
                {
```

```
                    k.push(node_cost);
                }
            }
        }
    }
    delete[] nodesAdded;
    return mst_tree_cost;
}


int main()
{
    adyacencias fromNode_0_in_graph_1 = { {1,1}, {2,2}, {1,3}, {1,4}, {2,5}, {1,6} };
    adyacencias fromNode_1_in_graph_1 = { {1,0}, {2,2}, {2,6} };
    adyacencias fromNode_2_in_graph_1 = { {2,0}, {2,1}, {1,3} };
    adyacencias fromNode_3_in_graph_1 = { {1,0}, {1,2}, {2,4} };
    adyacencias fromNode_4_in_graph_1 = { {1,0}, {2,3}, {2,5} };
    adyacencias fromNode_5_in_graph_1 = { {2,0}, {2,4}, {1,6} };
    adyacencias fromNode_6_in_graph_1 = { {1,0}, {2,2}, {1,5} };

    int num_of_nodes = 7; // Total Nodes (0 to 6)
    vector<adyacencias> primsgraph;
    primsgraph.resize(num_of_nodes);
    primsgraph[0] = fromNode_0_in_graph_1;
    primsgraph[1] = fromNode_1_in_graph_1;
    primsgraph[2] = fromNode_2_in_graph_1;
    primsgraph[3] = fromNode_3_in_graph_1;
    primsgraph[4] = fromNode_4_in_graph_1;
    primsgraph[5] = fromNode_5_in_graph_1;
    primsgraph[6] = fromNode_6_in_graph_1;

    // As we already know, we have to choose the source vertex,
    // so we start from the vertex 0 node.
    cout << "k-mst : " "" << PrimsMST(3, primsgraph, 3) << std::endl;
    return 0;
}
```

4.3. Resultado

4.3.1. Entrada

```
int main()
{
    //typedef vector<int> valNode;
    //typedef vector<valNode> adyacencias;
    adyacencias fromNode_0_in_graph_1 = { {1,1}, {2,2}, {1,3}, {1,4}, {2,5}, {1,6} };
    adyacencias fromNode_1_in_graph_1 = { {1,0}, {2,2}, {2,6} };
    adyacencias fromNode_2_in_graph_1 = { {2,0}, {2,1}, {1,3} };
    adyacencias fromNode_3_in_graph_1 = { {1,0}, {1,2}, {2,4} };
    adyacencias fromNode_4_in_graph_1 = { {1,0}, {2,3}, {2,5} };
    adyacencias fromNode_5_in_graph_1 = { {2,0}, {2,4}, {1,6} };
    adyacencias fromNode_6_in_graph_1 = { {1,0}, {2,2}, {1,5} };

    int num_of_nodes = 7; // Total Nodes (0 to 6)
    vector<adyacencias> primsgraph;

    primsgraph.resize(num_of_nodes);
    primsgraph[0] = fromNode_0_in_graph_1;
    primsgraph[1] = fromNode_1_in_graph_1;
    primsgraph[2] = fromNode_2_in_graph_1;
    primsgraph[3] = fromNode_3_in_graph_1;
    primsgraph[4] = fromNode_4_in_graph_1;
    primsgraph[5] = fromNode_5_in_graph_1;
    primsgraph[6] = fromNode_6_in_graph_1;

    // As we already know, we have to choose the source vertex,
    // so we start from the vertex 0 node.
    cout << "k-mst : " "" << PrimsMST(3, primsgraph, 3) << std::endl;
    return 0;
}
```

Figura 4: Entrada del algoritmo de fuerza bruta. Obtención propia.

4.3.2. Salida

```
Microsoft Visual Studio Debug Console
k-mst : 2

C:\Users\USER\source\repos\ADA\Debug\Prim_minimum.exe (process 26820) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close th
le when debugging stops.
Press any key to close this window . . .
```

Figura 5: Salida del algoritmo de fuerza bruta. Obtención propia.

5. Algoritmo aproximado

5.1. *Idea*

Según el artículo de Subham Datta[1], *Branch and Bound* es un algoritmo que optimiza otros algoritmos estableciendo limitaciones en el conjunto de respuestas posible. Es usado ampliamente en problemas NP-completos para hallar con facilidad computacional un resultado aproximado. Normalmente los *Upper Bounds* o límites superiores son condiciones que limitan el avance de la cantidad de respuestas, depende de la lógica del problema a tratar pero, en general, se toma un punto específico dentro del área de búsqueda de respuestas.

En caso del k-MST, se hacen varios tratamientos para que no tenga que buscar exhaustivamente la respuesta desde todos los nodos como origen. Primero se establecen nodos de partida que estén relacionados con aristas de peso muy bajo, luego se elige un valor específico que interrumpa la búsqueda al ser superado. Este es el límite superior. También se hace caso de una enumeración diferente, donde a los nodos se les da un valor precomputarizado de aristas / peso y con ese valor poder decidir qué valores añadir al grafo actual y cuáles no.

5.2. *Implementación*

```
#include <iostream>
#include <unordered_set>
#include <set>
#include <list>
#include <queue>
#include <bitset>
#include "grafos.h"
#include "abstract.h"
using namespace std;

typedef Graf<int, int> G;
//typedef unordered_set<Arista<G>> HashSet;

typedef bitset<4> BitSet;

void PrintMatrix(vector<vector<int>> mat) {
  for (int i = 0; i < mat.size(); i++) {
    for (int j = 0; j < mat[0].size(); j++) {
      cout << mat[i][j] << "\t";
```

```
    }
    cout << endl;
  }
}

/////////////APROXIMADO: PRIM OPTIMIZADO + BRANCH & BOUND ////////////////////


//KMST
class KMST {
private:
  vector<vector<Arista<G>>*> edgesFromNode;

  set<HashSet> visited;
  vector<int> minSum;
  int numNodes;
  int numEdges;
  int k = 0;
  int minWeight = INT_MAX;
  int kEdges;
  int limit;
  int abort;
  HashSet edges;
  bool limited;

public:
  //------------------------------------------------------------------

  bool hasNoCircle(BitSet used, int node1, int node2) {
    if (used[node1] && used[node2]) {
      return false;
    }
    return true;
  }
  //-----------------------------------------
  KMST(int numNodess, int numEdgess, HashSet edges, int k) {
    this->numNodes = numNodess;
    this->numEdges = numEdgess;
    this->edges = edges;
    this->k = k;
    this->kEdges = k - 1;
    edgesFromNode.resize(numNodes);
    minSum.resize(k);
```

```
    this->abort = 0;
    this->limited = true;

    //2*|V(G)|
    this->limit = 2 * numNodes * numNodes;


    // prioridad a los edges mas baratos
    priority_queue<Arista<G>>* min = new priority_queue<Arista<G>>;

    //lista adyacencia
    for (Arista<G> t : edges) {

      if (edgesFromNode[t.node1] == 0) {
        edgesFromNode[t.node1] = new vector<Arista<G>>(numNodes);
      }
      if (edgesFromNode[t.node2] == 0) {
        edgesFromNode[t.node2] = new vector<Arista<G>>(numNodes);
      }
      edgesFromNode[t.node1]->push_back(t);
      edgesFromNode[t.node2]->push_back(t);
      min->push(t);
    }

    minSum[0] = 0;
    for (int i = 1; i < k; i++) {
      minSum[i] = minSum[i - 1] + min->top().m_v;
      min->pop();
    }
  }

  //--------------------------------------------------------------

  void run() {
    constructMST();

  }

  //-----------------------------------------------------------------------------
```

```
  void constructMST() {
    vector<Arista<G>> aux(numNodes);
    priority_queue<Arista<G>> q(aux.begin(),aux.end());
    int t;

    //  a adir  suma de nodos en orden inverso
    for (int i = 0; i < numNodes; i++) {
      t = getBestEdge(i);
      if (t != INT_MAX) {
        Arista<G> a(t);
        a.node1 = i;
        a.node2 = -1;
        Arista<G> ari(i, -1, t);
        q.push(ari);
      }
    }


    priority_queue<Arista<G>> q_prim = q;
    priority_queue<Arista<G>> q_limited = q;

    //limite superior
    while (!q_prim.empty()) {
      vector<Arista<G>> aux(k);
      vector<Arista<G>> aux2(numEdges);
      firstEstimate(*new HashSet(aux.begin(), aux.end()),
   q_prim.top().node2, 0,
          *new priority_queue<Arista<G>>(aux2.begin(), aux2.
   end()), *new BitSet(numNodes), 0);
      q_prim.pop();
    }
    while (!q_limited.empty()) {
      HashSet n;

      addNodes(n, q_limited.top().node2, 0, q,
          *new BitSet(numNodes), 0);
      q_limited.pop();
      abort = 0;
    }

    visited.clear();
    limited = false;
    limit = INT_MAX;

    while (!q.empty()) {
```

```
    HashSet n;
    priority_queue<Arista<G>> p;
    addNodes(n, q.top().node2, 0, p, *new BitSet(numNodes),
  0);
    q.pop();
  }
  cout<<"finish"<<endl;

}


//---------------------------------------------------
int getBestArista(int node, vector<vector<int>> mat) {
  int ret = 0;
  for (int e : mat[node]) {
    ret += e;
  }
  return ret * -1;

}

//
 ---------------------------------------------------------------

bool find(priority_queue<Arista<G>> e, const int& val)
 const
{
  while (!e.empty()) {
    if (e.top().m_v == val) return true;
    e.pop();
  }
  return false;
}

//
 ------------------------------------------------------------------------

bool find(set<HashSet> e, HashSet adj) const
{
  for (auto& a : e) {
    if (a == adj)
      return true;
  }
  return false;
}
```

```

// ---------------------------------------------------------------------- falta acomodar*
void addToQueue(priority_queue<Arista<G>> e, int node, BitSet used, int w,
  int numEdges) {

  for (Arista<G> ite : (*edgesFromNode[node])) {
    if (!used[node == ite.node1 ? ite.node2 : ite.node2]
      && w + ite.m_v + minSum[kEdges - numEdges - 1] < minWeight
      && !find(e, ite.m_v)) {
      e.push(ite);
    }
  }
}

// ----------------------------------------------------------------------
void firstEstimate(HashSet e, int node, int cweight,
  priority_queue<Arista<G>> p, BitSet used, int numAristas) {
  Arista<G> t;
  int w, newNode;
  bool abort = false, wasEmpty, solutionFound;


  addToQueue(p, node, used, cweight, numAristas);

  while (!p.empty() && !abort) {
    t = p.top();
    p.pop();

    if (t.m_v >= minWeight) {
      edgesFromNode[t.node1]->erase((edgesFromNode[t.node1]->begin() + t.node2));
      edgesFromNode[t.node2]->erase((edgesFromNode[t.node2]->begin() + t.node1));

    }
    else {
      w = cweight + t.m_v;

```

```
      if (hasNoCircle(used, t.node1, t.node2)) {

        abort = true;

        if (used[t.node1]) {

          newNode = t.node2;
          node = t.node1;
        }
        else {
          newNode = t.node1;
          node = t.node2;
        }

        //  a adir  arista
        e.insert(t);

        wasEmpty = false;
        solutionFound = false;

        if (used.none()) {
          // primera arista
          used.set(newNode);
          used.set(node);
          wasEmpty = true;
        }
        else {
          used.set(newNode);
        }

        int size = used.count();

        //actualizar solucion
        if (size == k && w < minWeight) {
          minWeight = w;
          AbstractKMST o; //implementR UPDATEsolution
          o.setSolution(w, e);
        }
        else if (size < k) {

          firstEstimate(e, newNode, w, p, used, numAristas + 1);
        }

```

```
        if (!solutionFound) {
          used.reset(newNode);
          if (wasEmpty) {
            used.reset(node);
          }
        }
      }
    }
  }
}

//
 ------------------------------------------------------------------

int getBestEdge(int node) {
  int ret = 0;
  for (Arista<G> e : *edgesFromNode[node]) {
    ret += e.m_v;
  }
  return ret * -1;

}


//
 ------------------------------------------------------------------

void addNodes(HashSet e, int node, int cweight,
 priority_queue<Arista<G>> p, BitSet used, int numAristas)
 {


  if (limited)
    abort++;

  Arista<G> t;
  vector<Arista<G>> aux(2 * k);
  HashSet* temp = new HashSet(aux.begin(), aux.end());
  int w, newNode, size;
  bool wasEmpty, solutionFound;

  // clone
  if (!p.empty()) {
    p = *new priority_queue<Arista<G>>(p);
  }
```

```
    else {
      p = *new priority_queue<Arista<G>>();
    }

     if (used != NULL) {
      used = used;
     }

    if (!e.empty()) {
      temp->insert(e.begin(), e.end());
    }

    // expand node
    addToQueue(p, node, used, cweight, numAristas);

    while (!p.empty() && abort < limit) {
      t = p.top();
      p.pop();
      w = cweight + t.m_v;


      if (w + minSum[kEdges - numAristas - 1] < minWeight
        && !find(visited, *temp)) {


        if (hasNoCircle(used, t.node1, t.node2)) {
          if (used[t.node1]) {

            newNode = t.node2;
            node = t.node1;
          }
          else {
            newNode = t.node1;
            node = t.node2;
          }

          temp->insert(t);

          wasEmpty = false;
          solutionFound = false;
          if (used.none()) {
            // first edge
            used.set(newNode);
            used.set(node);
            wasEmpty = true;
```

```
        }
        else {
          used.set(newNode);
        }


        size = used.count();

        if (size == k) {
          // nueva solucion encontrada
          updateSolution(*temp, w);
          solutionFound = true;
          abort = 0;
        }
        else {

          addNodes(*temp, newNode, w, p, used, numAristas + 1);

          if (size == 2) {
            visited.insert(*temp);
          }

          // revert to starting solution
          vector<Arista<G>> helper(k); //esto solo es para poder inicializar el set con un tama o especifico
          temp = new HashSet(helper.begin(), helper.end());
          if (!e.empty()) {
            temp->insert(e.begin(), e.end());
          }
        }
        // clear nodes
        if (!solutionFound) {
          used.reset(newNode);
          if (wasEmpty) {
            used.reset(node);
          }
        }
      }
    }
      //si encuentra la solucion
    else {
      cout<<"solucion encontrada"<<endl;
      break;
    }
```

```
    }
  }

  // ------------------------------------------------------------------

  void updateSolution(HashSet minSet, int min) {
    minWeight = min;
    AbstractKMST a;
    a.setSolution(min, minSet);
    cout<<min<<endl;

  }
  //-----------------------------------------
};
int main()
{
  G LOL;
  LOL.insertNode(1);
  LOL.insertNode(2);
  LOL.insertNode(3);
  LOL.insertNode(4);
  LOL.insertNode(5);
  LOL.insertNode(6);
  LOL.insertArista(1, 2, 3, false);
  LOL.insertArista(5, 4, 4, false);
  LOL.insertArista(5, 3, 3, false);
  LOL.insertArista(4, 2, 1, false);
  LOL.insertArista(4, 6, 6, false);
  vector<vector<int>> a = AdjacencyFromGraph(LOL);
  PrintMatrix(a);

  HashSet ar = HashFromGraph(LOL);
  for (auto& ari : ar){
    cout<<ari.m_nodos[0]->id<<" "<<ari.m_v<<endl;
  }

}
```

6. Aplicaciones

6.1. Network Design

Una de sus aplicaciones más conocidas respecto a problemas de conexión, por ejemplo una compañia de celular, que tiene distintos precios por diferentes pares de ciudades, el problema esta en construir la red de menor costo posible dado estas ciudades. O bien para una agencia de viajes, y se busca determinar el alcance y costo de la aerolínea.

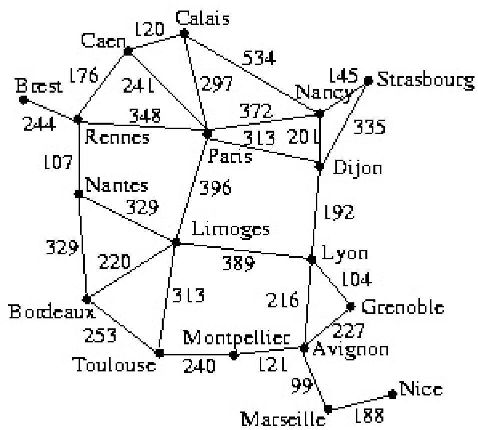

Figura 6: Grafo de ciudades. Fuente: https://people.cs.georgetown.edu/ maloof/-cosc270.f17/p2.html

Referencias

[1] Datta, S. (10 de Octubre de 2020). baeldung.com. Obtenido de https://www.baeldung.com/cs/branch-and-bound

[2] Gupta, S. (Junio de 2022). geeksforgeeks. Obtenido de https://www.geeksforgeeks.org/steiner-tree/

[3] Matt Elder, S. C. (2007). CS880: Approximation Algorithms. Obtenido de https://pages.cs.wisc.edu/ shuchi/courses/880-S07/scribe-notes/lecture26-2.pdf

[4] R. Ravi, R. S. (12 de Julio de 2006). Spanning Trees—Short or Small. Obtenido de SIAM (Society for Industrial and Applied Mathematics: https://epubs.siam.org/doi/pdf/10.1137/S0895480194266331

[5] Wikipedia. (Junio de 2022). Wikipedia. Obtenido de https://en.wikipedia.org/wiki/K-minimumspanningtree