k-subminimum tree of G

Víctor Emiliano Cruz Hernández

27 de abril de 2023

1. Introducción

El problema del k-subárbol-mínimo es un problema de optimización combinatoria en el que se busca encontrar un subconjunto de k aristas en una gráfica tal que el subgrafo formado por esas k aristas tenga el menor peso posible.

Algunas aplicaciones del problema se encuentran en la optimización de redes de comunicaciones y en la planificación de redes eléctricas. Es un problema NP-duro, lo que significa que no existe un algoritmo eficiente que pueda resolver el problema para todos los casos de entrada. Se han desarrollado diferentes enfoques heurísticos para resolver este problema de manera aproximada, en este documento nos basamos en una técnica híbrida.

2. Heurística

2.1. Vecindario

Sea $g = (V, E) = (V(G), E(G))$ nuestra gráfica de entrada. Definimos el espacio de búsqueda de acuerdo a las siguientes definiciones:

Definition 2.1 (T_k)**.** T_k es un k-subárbol de $G \Leftrightarrow T_k \in G$ y $k \leq |V(G)| - 1$.

Definition 2.2 ($V_{NH}(T_k)$)**.** Definimos a $V_{NH}(T_k)$ como el conjunto de vértices de $V(G)$ que no están en $T - k$ pero son adyacentes a él.

$$V_{NH}(T_k) = \{v \in V(G) - V(T_k) | \exists v' \in V(T_k)(\{v, v'\} \in E(G)\}$$

Definition 2.3 (El Vecindario de T_k: $NH(T_k)$)**.** Sea $T_k = (V(T_k), E(T_k))$ para el cual obtenemos su $V_{NH}(T_k)$.

Creamos una solución local mínima T_k^{NH} de T_k añadiendo un vértice $v_{in} \in V_{NH}(T_k)$ obtenido aleatoriamente a T. Posteriormente con nuestro algoritmo de búsqueda local eliminaremos algún $v_{out} \in V(T_k)$ escogido en base a los pesos de sus aristas.

Naturalmente seleccionamos la siguiente solución a iterar como el $T_k^{NH} \in NH(T_k^{cur})$ que tenga la mejor función objetivo de todas. A este lo denotamos

como $T_k^{NH_{best}}$ que cumple con:

$$T_k^{NH_{best}} = min_{T_k^{NH} \in NH(T_k^{cur})}\{f(T_k^{NH})\} \tag{1}$$

2.2. Parámetros para la búsqueda tabú

Para la búsqueda tabú vamos a definir un conjunto de valores tt_{min}, tt_{max}, tt_{inc} y nic_{max}. Vamos a definir un intervalo nic_{int} (*Periodo De Estancamiento*) y dos listas: $TabuInList$, $TabuOutList$ para evitar ciclarnos entre el conjunto de soluciones $NH(T_k^{cur})$, mientras ejecutamos el **algoritmo de búsqueda local**.

$$tt_{min} := min\Big\{\Big\lfloor\frac{|V|}{20}\Big\rfloor, \frac{|V|-k}{4}, \frac{k}{4}\Big\}$$

Definition 2.4 (tt_{max})**.** Defininimos a tt_{max} como:

$$tt_{max} := \Big\lfloor\frac{|V|}{5}\Big\rfloor$$

Definition 2.5 (tt_{inc})**.** Defininimos a tt_{inc} como:

$$tt_{inc} := \Big\lfloor\frac{tt_{max} - tt_{min}}{10}\Big\rfloor + 1$$

Definition 2.6 (nic_{max})**.** Defininimos a nic_{max} como:

$$nic_{max} := max\{tt_{inc}, 100\}$$

Definition 2.7 (*Periodo de Estancamiento*: nic_{int})**.** Dada una mejor solución encontrada $T_k^{NH_{best}}$, definimos a nic_{int} como el periodo en el que no hemos actualizado esa solución.

Definition 2.8 ($TabuInList$)**.** Una lista que contiene a los índices de las aristas removidas.

Definition 2.9 ($TabuOutList$)**.** Una lista que contiene a los índices de las listas añadidas.

Inicialmente vamos a declarar tt_{ten} asignándole el valor de tt_{min}. Cuando $nic_{int} > nic_{max}$ actualizamos tt_{ten} como $tl_{ten} \leftarrow tt_{ten} + tt_{nic}$.

2.3. Algoritmo de búsqueda local basada en búsqueda tabú

Sea $G = (V(G), E(G))$ nuestra gráfica de entrada y una $k \leq |V(G)| - 1$.

2.3.1. Generación de la solución inicial T_k^{cur}

Seleccionamos un nodo al azar de $V(G)$ y aplicamos el algoritmo `Prim` hasta que terminemos de construir un k-subárbol, llamémosle T_k^{cur}. Entonces definimos a T_k^{cur} como nuestra *solución inicial* y nuestra *solución actual*

2.3.2. Inicializamos los parámetros de búsqueda tabú

Inicializamos los parámetros que se describen en la sección de Parámetros para la búsqueda tabú.

2.3.3. Búsqueda basada en búsqueda tabú

Paso 1. Inicializamos la lista de un nodo detectado:
Inicializamos V_{in} como $V_{in} \leftarrow V_{NH}(T_k^{cur})$.

Paso 2. Decisión de eliminar un nodo:
Si $V_{in} = \varnothing$ terminamos el algoritmo. En otro caso vamos al **Paso 2.1**.

Paso 2.1 Encontrar el vértice v_{in}:

Definition 2.10 (El vértice v_{in})**.**

$$v_{in} := min_{v \in V_{in}} \left\{ \frac{\sum_{v' \in V(T_k^{cur})} w(e)}{d(v)} \Big| e = (v, v') \right\}$$

Vamos a encontrar v_{in} como el mínimo vértice $v \in V_{in}$ tal que la media del peso de las aristas que conectan con nuestra trayectoria actual T_k^{cur}, sea el menor.

En la definición anterior lo hacemos de la siguiente forma. Para todo vértice $v \in V_{in}$:

1. Vamos a calcular el número de aristas que conectan v con algún otro vértice v' de nuestra trayectoria actual T_k^{cur} y llamaremos a este número $d(v)$.
2. Posteriormente, vamos a sumar el peso de cada una de las aristas $w(e) = w((v, v'))$ que la conectan con la trayectoria actual en $\sum_{v' \in V(T_k^{cur})} w(e)$, llamémosle s.
3. Calculamos la media del peso de las aristas en $\frac{s}{d(v)}$
4. Obtenemos a v_{in} como aquel con un valor mínimo de $\frac{s}{d(v)}$ de entre todos los vértices de V_{in}.

Establecemos a V_{in} como

$$V_{in} \leftarrow V_{in} - \{v_{in}\}$$

Procedemos al paso 2.2.

Paso 2.2 Encontrar la arista e_{min_1}:

Definition 2.11 (El conjunto de aristas E_{in_1})**.**

$$E_{in_1} := \{(v, v_{in}) | v \in V(T_k^{cur})\}$$

Definition 2.12 (La arista e_{min_1})**.**

$$e_{min_1} := min_{e \in E_{in_1}}\{w(e)\}$$

Observamos que pudimos haber construido el conjunto de aristas E_{in_1} indirectamente en el paso 2.1 al encontrar v_{in}, pues la suma s correspondiente al vértice v_{in} justamente itera sobre el conjunto de aristas que queremos encontrar.

De cualquier manera, para construir E_{in_1} tomamos nuestro vértice v_{in} y para cada vértice $v \in V(T_k^{cur})$, la arista (v_{in}, v) se añade a E_{in_1}. Tenemos que e_{min_1} es simplemente la arista de menor peso que conecta a v_{in} con nuestra trayectoria T_k^{cur}.
Convenientemente para este paso vamos a modificar E_{in_1} como:

$$E_{in_1} \leftarrow E_{in_1} - \{e_{min_1}\}$$

Y ya que hayamos encontrado v_{in} y e_{min_1} creamos un árbol T_{k+1}^{NH} de la siguiente manera:

$$T_{k+1}^{NH} \leftarrow (V(T_k^{cur}) \cup \{v_{in}\}, E(T_k^{cur}) \cup \{e_{min_1}\})$$

Lo que ahora hemos hecho es crear un árbol de $k+1$ vértices exactamente igual a nuestro árbol T_k^{cur} pero añadiendo el vértice v_{in} con su respectiva arista de menor peso que la conecta con nuestro árbol T_k^{cur}.

No hemos acabado de manipular a nuestro nuevo árbol T_{k+1}^{NH}. Procedemos al paso 2.3.

Paso 2.3 Volver a encontrar la arista e_{min_1}:

Simplemente quitamos de E_{in_1} a e_{min_1} de forma que obtenemos la segunda arista de menor peso que conecta v_{in} con nuestro árbol T_k^{cur}.
Convenientemente para este paso vamos a modificar E_{in_1} como:

$$E_{in_1} \leftarrow E_{in_1} - \{e_{min_1}\}$$

Y además modificamos a T_{k+1}^{NH} añadiéndole a e_{min_1}:

$$T_{k+1}^{NH} \leftarrow (V(T_{k+1}^{NH}), E(T_{k+1}^{NH}) \cup \{e_{min_1}\})$$

Observamos que ahora T_{k+1}^{NH} ya no es un árbol pues al añadir e_{min_1} una vez para después volver a encontrar e_{min_1} y volver a añadirlo, ambas aristas conectan a v_{in} en T_k^{cur} y entonces hemos creado un ciclo en T_{k+1}^{NH}. Con este conocimiento procedemos al paso 2.4.

Paso 2.4 Encontrar la arista e_{max}:
Sea E_{loop} el conjunto de aristas que forman el ciclo en T_{k+1}^{NH} construido previamente. Buscamos a e_{max} como:

Definition 2.13 (La arista e_{max})**.**

$$e_{max} := max_{e \in E_{loop}}\{w(e)\}$$

Modificamos a T_{k+1}^{NH}, para que vuelva a ser un árbol de la siguiente forma:

$$T_{k+1}^{NH} \leftarrow (V(T_{k+1}^{NH}), E(T_{k+1}^{NH}) - \{e_{max}\})$$

Paso 2.5 Iterar hasta asegurarse que no hay mejor trayectoria que conecte a v_{in} con T_k^{cur}:
Iteramos desde el paso 2.3 mientras que $E_{in_1} \neq \varnothing$. Observamos que a medida que iteramos, vamos a ir conectando a el vértice v_{in} con las distintas posibles aristas que lo conectan con la trayectoria T_k^{cur} intentando construir una $k+1$-trayectoria sobre nuestra k-trayectoria actual T_k^{cur} de forma que tenga el menor costo posible.

Cuando $E_{in_1} = \varnothing$ entonces declaramos nuestro conjunto de vértices V_{out} como:

$$V_{out} \leftarrow V(T_k^{cur})$$

Y a partir de este momento continuamos en el paso 3.

Paso 3. Decisión de un nodo añadido para construir T_k^{NH}:
Si $V_{out} = \varnothing$ regresamos al Paso 2. En otro caso vamos al paso 3.1.

Paso 3.1 Encontrar al vértice v_{out}:

Definition 2.14 (El vértice v_{out})**.**

$$v_{out} := max_{v \in V_{out}} \left\{ \frac{\sum_{v' \in V(T_k^{cur})} w(e)}{d(v)} \Big| e = (v, v') \right\}$$

La construcción de la definición anterior es similar a v_{in}, en lugar de obtener una mínima arista, obtenemos una máxima, pero de todas formas se explica:
Vamos a encontrar v_{out} como el máximo vértice $v \in V_{out}$ tal que la media del peso de las aristas que conectan con nuestra trayectoria actual T_k^{cur}, sea máxima.
Para todo vértice $v \in V_{out}$:

1. Vamos a calcular el número de aristas que conectan v con algún otro vértice v' de nuestra trayectoria actual T_k^{cur} y llamaremos a este número $d(v)$.
2. Posteriormente, vamos a sumar el peso de cada una de las aristas $w(e) = w((v, v'))$ que la conectan con la trayectoria actual en $\sum_{v' \in V(T_k^{cur})} w(e)$, llamémosle s'.
3. Calculamos la media del peso de las aristas en $\frac{s'}{d(v)}$
4. Obtenemos a v_{in} como aquel con un valor máximo de $\frac{s'}{d(v)}$ de entre todos los vértices de V_{in}.

Después establecemos a V_{out} como

$$V_{out} \leftarrow V_{out} - \{v_{out}\}$$

Continuamos en el paso 3.2.

Paso 3.1 Encontrar a la arista e_{min}^{out}:

Definition 2.15 (La arista e_{min}^{out})**.**

$$e_{min}^{out} := min_{e \in \{(v_{out}, v') | v' \in T_k^{cur}\}} \{w(e)\}$$

Ahora lo que estamos buscando es encontrar la arista de menor peso que conecta nuestro vértice v_{out} a nuestra trayectoria T_k^{cur}. De nuevo, el conjunto donde vamos a encontrar v_{out} nos va a dar e_{min}^{out} pues esta forma parte de la suma s'.

Ahora si para e_{min}^{out} dado y nuestro T_{k+1}^{NH} construido previamente sucede que:

$$f(T_k^{NH_{best}}) < \left(\sum_{e \in E(T_{k+1}^{NH})} w(e) \right) - w(e_{min}^{out})$$

vamos a regresar directamente al paso 3. De otra forma vamos a ir al paso 3.3.

Paso 3.3 Las componentes conexas S_r que forman al árbol T_k^{NH}:

Eliminamos v_{in} de T_{k+1}^{NH} de forma que obtenemos un bosque $(S_1, \ldots, S_r)$. Aunque obtenemos un bosque, vamos a tomar este bosque como nuesto candidato al árbol T_k^{NH} para el cual nos preocuparemos por construir en lo que resta del paso 3, si es que merece la pena. Procedemos ahora a encontrar el conjunto de aristas E_{in_2}:

Definition 2.16 (Las aristas E_{in_2})**.** Sea el bosque $(S_1, \ldots, S_r)$ obtenido de T_{k+1}^{NH} al eliminar v_{in}. Definimos a E_{in_2} como:

$$E_{in_2} = \{(v_i, v_j) | v_i \in S_k, v_j \in S_l, k \neq l\}$$

Pasamos al paso 3.4

Paso 3.4 Encontrar a la arista e_{min_2}:

Definition 2.17 (La arista e_{min_2})**.** Sea nuestro conjunto de aristas E_{in_2}. La arista e_{min_2} es la de menor peso de entre ellas:

$$e_{min_2} = min_{e \in E_{in_2}} \{w(e)\}$$

Si sucede que para e_{min_2}

$$f(T_k^{NH_{best}}) < w(e_{min_2}) + \sum_{e \in E(T_k^{NH})} w(e) \tag{2}$$

Entonces regresamos al paso 3, de otra forma vamos al paso 3.5

Paso 3.5 Unir la arista e_{min_2} con el árbol no conexo T_k^{NH}:
Si agregar e_{min_2} a T_k^{NH} no nos genera un ciclo, entonces $E(T_k^{NH}) \leftarrow E(T_k^{NH}) \cup e_{min_2}$ en ese y cualquier otro caso eliminamos de E_{in_2}. Procedemos al paso 3.6

Paso 3.5 Verificar si T_k^{NH} ya es un árbol:
Si T_k^{NH} es un árbol, entonces actualizamos $T_k^{NH_{best}}$ a T_k^{NH} y regresamos al paso 3. Si no es un árbol regresamos al paso 3.4

3. Implementación

La implementación se sigue con el siguiente pseudocódigo:

Algorithm 1: BusquedaTabu

Data: K_n, por definición, operamos bajo una gráfica geométrica completa
Result: $T_k \in K_n, k \leq n-1$

```
InicializarParametrosYListas();
T_k^cur ← GenerarSolucionInicial();
T_k^gb ← T_k^cur;
T_k^rb ← T_k^cur;
V_in ← V_NH(T_k^cur);
V_out ← ∅;
cont ← 0;
while tt_ten < tt_max and cont < maximoNumeroIteraciones do
    cont ← cont + 1;
    if V_out = ∅ then
        while V_out = ∅ do
            if V_in = ∅ then
                return
            else
                T_{k+1}^NH ← DecisionDeUnNodoEliminado(T_k^cur, V_in, V_out);
            end
        end
    else
        v_out ← max_{v∈V_out} { (Σ_{v'∈V(T_k^cur)} w(e)) / d(v) | e = (v, v') };
        V_out ← V_out − {v_out};
        e_min^out ← min_{e∈{(v_out,v')|v'∈T_k^cur}} {w(e)};
        if f(T_k^{NH_best}) < ( Σ_{e∈E(T_{k+1}^NH)} w(e) ) − w(e_min^out) then
            continue;
        end
        T_k^NH = (S_1, ..., S_r) ←
         obtieneComponentesConexasAlEliminarVertice(v_in, T_{k+1}^NH);
        E_in2 ← {(v_i, v_j)|v_i ∈ S_k, v_j ∈ S_l, k ≠ l};
        do
            e_min2 ← min_{e∈E_in2} {w(e)};
            if f(T_k^{NH_best}) < w(e_min2) + Σ_{e∈E(T_k^NH)} w(e) then
                break;
            end
            if No hay ciclo en e_min2 ∪ T_k^NH then
                E(T_k^NH) ← E(T_k^NH) ∪ e_min2;
            end
            E_in2 ← E_in2 − {e_min2};
            if T_k^NH es un árbol then
                f(T_k^{NH_best}) ← f(T_k^cur);
            end
        while T_k^NH no es un árbol;
    end
end
```

Algorithm 2: DecisionDeUnNodoEliminado

Data: $T_k^{cur}, V_{in}, V_{out}$
Result: T_{k+1}^{NH}
$T_{k+1}^{NH} \leftarrow \varnothing$;
$v_{in} \leftarrow min_{v \in V_{in}} \left\{ \frac{\sum_{v' \in V(T_k^{cur})} w(e)}{d(v)} \middle| e = (v, v') \right\}$;
$V_{in} \leftarrow V_{in} - \{v_{in}\}$;
$E_{in_1} \leftarrow \{(v, v_{in}) | v \in V(T_k^{cur})\}$;
$e_{min_1} \leftarrow min_{e \in E_{in_1}} \{w(e)\}$;
$T_{k+1}^{NH} \leftarrow (V(T_k^{cur}) \cup \{v_{in}\}, E(T_k^{cur}) \cup \{e_{min_1}\})$;
$E_{in_1} \leftarrow E_{in_1} - \{e_{min_1}\}$;
while $E_{in_1} \neq \varnothing$ **do**
  $e_{min_1} \leftarrow min_{e \in E_{in_1}} \{w(e)\}$;
  $T_{k+1}^{NH} \leftarrow (V(T_{k+1}^{NH}), E(T_{k+1}^{NH}) \cup \{e_{min_1}\})$;
  $E_{in_1} \leftarrow E_{in_1} - \{e_{min_1}\}$;
  $E_{loop} \leftarrow$ `DetectaCiclo(`T_{k+1}^{NH}`)` ;
  $e_{max} \leftarrow max_{e \in E_{loop}} \{w(e)\}$;
  $T_{k+1}^{NH} \leftarrow (V(T_{k+1}^{NH}), E(T_{k+1}^{NH}) - \{e_{max}\})$;
end
$V_{out} \leftarrow V(T_k^{cur})$;
return T_{k+1}^{NH};

Algorithm 3: GenerarSolucionInicial

$v_{initial} \leftarrow$ `RandomVertex()`;
Aplicamos `Prim` *hasta generar un k-arbol* ;
$T_k^{cur} \leftarrow$ `Prim(`$v_{initial}, k$`)`;
return T_k^{cur};

Algorithm 4: InicializarParametrosYListas

$tt_{min} \leftarrow min\left\{ \left\lfloor \frac{|V|}{20} \right\rfloor, \frac{|V|-k}{4}, \frac{k}{4} \right\}$;
$tt_{max} \leftarrow \left\lfloor \frac{|V|}{5} \right\rfloor$;
$tt_{inc} \leftarrow \left\lfloor \frac{tt_{max} - tt_{min}}{10} \right\rfloor + 1$;
$nic_{max} \leftarrow max\{tt_{inc}, 100\}$;
$tt_{ten} \leftarrow tt_{min}$;
$nic_{int} \leftarrow 0$;
$InList \leftarrow []$;
$OutList \leftarrow []$;
$maximoNumeroIteraciones = 100000$;

4. Resultados

El sistema funciona, encuentra soluciones reales dentro de la gráfica y es relativamente rápido. De acuerdo a la heurística, no debería ciclarse alguna sección de la implementación, aún así se añadió un corto circuito del primer while del algoritmo `BusquedaTabu` pues este se ciclaba. La implementación no es limpia y no aprovecha los recursos que Rust ofrece. Tanto los algoritmos como las estructuras de datos no son los más óptimos en este problema.

Aunque nuestra heurística se basara fuertemente en el uso de las listas `InList` y `OutList`, estas no figuraban una participación transparente en el artículo de la heurísitca, por los que los adapté de una segunda fuente mencionada en las referencias del artículo.

No se implementó la segunda parte de la heurística, la parte de exploración con hormigaas si bien es parte de la heurística, no forma un fuerte acoplamiento con la búsqueda tabú por lo que su implementación puede hacerse en un futuro.

Referencias

[1] Hideki Katagiri, Tomohiro Hayashida, Ichiro Nishizaki, and Jun Ishimatsu. A hybrid algorithm based on tabu search and ant colony optimization for k-minimum spanning tree problems. In *PLecture Notes in Artificial Intelligence*, pages 315–326, Awaji Island, Japan, 2009.

[2] M. J. Blum, C.; Blesa. *New metaheuristic approaches for the edge-weighted k-cardinaity tree problem.* Univerrsitat Plitecnica de Catalunya, Barcelona, Spain, 2003.

[3] J. Blandy, J.; Orendorff. *Programming Rust.* O'Reilly, 1005 Gravenstein Highway North, Sebastopol, 2017.