5. ***Maximizing support***

Consider an undirected graph $G = (V, E)$. We say that a set $S \subseteq V$ supports a vertex $u \in V$ (or the u is supported by S) if u and all its neighbors in G belong to S.

The Max Restricted Support problem is defined as follows: Given an undirected graph $G = (V, E)$ and an integer s, $1 \leq s \leq |V|$, find a subset $S \subseteq V$ with $|S| = s$ such that the number of vertices supported by S is maximum over all sets with s vertices.

(b) Provide a FPT algorithm for the optimization problem parameterized by the treewidth of G.

- T és un nice tree decomposition de G
- Sigui $u \in V(T)$, P_u és una taula $P_u(n, S, C) = (p, X)$ tal que $S \subseteq X_u$, $C \subseteq X_u$, $s \geq (n + |(S \cup Q)|)$.
- Definim $M_u = \{S: S \subseteq X_u, \wedge s \geq |S|\}$

funció Hcol(G,T)

for $u \in V(T)$ **(iterats en postordre)**

if u és StartNode

for $S \in M_u$, $P_u(0, S, \{\}) = (0, \{\})$

if u és IntroduceNode

sigui v l'únic fill de u

$\{x\} = X_u - X_v$

for $P_v(n, S, C)$

$P_u(n, S, C) = P_v(n, S, C)$(copia)

$P_u(n, (S \cup \{x\}), C) = P_v(n, S, C)$

if u és ForgetNode

sigui v l'únic fill de u

$\{x\} = X_v - X_u$

$AD = \{i \mid i \in X_u \wedge (\exists (i, x) \in G[X_u])\}$

for $S \subseteq X_u, C \subseteq X_u, n \leq s$

$P_u(n, S, C) = max_p(P_v(n, S, C \setminus AD), P_v(n - 1, S \cup \{x\}, C), P_v(n, S \cup \{x\}, C \setminus (\{x\} \cup AD)))$

Esta linea esta regular mejor mirad el texto

if u és JoinNode

siguin $v_1 i\ v_2$els fills de u

for $S \subseteq X_u,\ C \subseteq X_u$

for $n \leq s$

$$P = \{(p_1 + p_2, Q_1 \cup Q_2) |\ P_{v1}(n_1, S, C) = (p_1, Q) \wedge P_{v2}(n_2, S, C) = (p_2, Q_2) \wedge n_1 + n_2 = n\}$$

$$P_u(n, S, S') = max_p(J)$$

borrar P_{v1} i P_{v2}

return $P_r(s, \emptyset, \emptyset)$

fi funció

Intento de explicar el algoritmo en palabras

Para parametrizar por treewidth partimos de que tenemos una rooted nice tree decomposition (T, X) del grafo G.

Para cada nodo v ∈ V(T) tenemos una tabla$P_v(n, S, C) = (p, Q)$ por cada S ⊆ X_v y por cada C ⊆ X_v. Dicho de otra manera, dado un nodo v del nice tree decomposition, tenemos una entrada de la tabla por cada posible permutación de incluir o no un vértice en el supporting set. Explicaremos más adelante el significado de C, pero de momento destacar que tenemos también entradas separadas que tienen el mismo S pero C distintos.

Esta entrada es una tupla con varios elementos (n, S, C). En S tenemos todos los vértices de X_v que esa entrada asigna al supporting set. Estos son los vértices que estamos mirando por brute force si vale más la pena incluir o no en nuestro supporting set. Y como no sabemos con certeza si todos sus vecinos van a ser parte del supporting set solucion, de momento no sabemos si añadirán o no “puntuación” a la solución.

En C tenemos vértices que pertenecen a v (no hemos olvidado aún), pero que en algún momento uno de sus adyacentes vértices ha sido decidido que no va a estar en el supporting set. Podemos enteder C como el set de vértices en v que no están soportados (no añaden “puntuación” de ser añadidos a S). Vulgarmente podemos pensar en ellos como nodos “cojos”.

n es la cantidad de elementos de Q.

Sobre la salida, queda definida como la tupla (p, Q), donde p es la “puntuación”, es decir el número de vértices de G que quedan soportados por el set S ∪ Q, pero sin contar los nodos de S, ya que no es definitivo que estos vayan a estar soportados (un futuro introduce

los puede dejar cojos). Entonces podemos ver que Q son los elementos que ya no aparecen en X_v pero que debemos recordar que pertenecen al supporting set que estamos construyendo, y que entonces sabemos exactamente si añaden 1 o no a la puntuación.

Sin querer detallar más en estos conjuntos por ahora, simplemente queremos introducirlos para poder ir detallando las operaciones que caracterizan los algoritmos parametrizados por treewith. Esperamos que las operaciones aclaren las dudas de las definiciones anteriores.

Start node:

Simplemente empezamos con un primer vértice, así que debemos crear las dos únicas entradas de tablas posibles. Incluir o no el vértice inicial en S.

Introduce:

Definimos el nodo del treewith "padre" como X_v, al que hemos introducido el vértice x, y su nodo hijo X_u que era el de antes de introducir x.

Por todas las entradas de X_u podemos ahora pensar que debemos elegir si añadir o no el vértice x. Por eso duplicamos las tablas, y por cada entrada ahora existe una idéntica que no ha introducido u (que es la misma entrada sin alterar nada). Y otra entrada nueva que sobre la entrada antigua ha añadido u al supporting set. Aquí no hace falta recalcular la puntuación de ninguna tabla, ya que la "puntuación" es sobre Q, no sobre S.

Aquí cabe destacar que se dará por trivial de ahora en adelante que el algoritmo al crear entradas de tabla puede comprobar si n es mayor al deseado y automáticamente descartar la tabla.

Forget

Esta operación es la que se encarga de "hacer commit" de las "decisiones" que se han ido tomando con los introduce. Podemos pensar que dada una entrada P, esta ha ido "decidiendo" si incluir los vértices que se han ido añadiendo. Y ahora toca hacer forget de cierto vértice x. Entonces imaginemos dos entradas que han tomado decisiones varias, pero se distinguen porque una incluye x y la otra no. Llamaremos a estas entradas P_x y P_{no_x} respectivamente.

Forget de x en la entrada P_x
Si el vértice x no era "cojo", es decir que no pertenece a C, entonces el vértice x va a no ser cojo durante el resto de la construcción, ya que una vez que una entrada de la tabla "decide" incluir en S cierto vértice, luego ya no hay marcha atrás. Entonces podemos estar seguros de que añadimos x a Q y que el estado de "cojo" o "no cojo" para esa x en esa entrada de la tabla ya no se va a alterar. Si x no era cojo, lo seguirá sin ser tras el forget, y luego ya es imposible alterar su condición de cojo. Lo mismo si era cojo, en que ya es imposible hacer que "deje de ser cojo". Para cada caso sumaremos 1 o no a la puntuación.

Forget de x en la entrada P_{no_x}

Entonces significa que cojo o no, x no ha sido “decidido” que pase a ser de la solución que está construyendo esa entrada de la tabla. Esto tiene consecuencias para el resto de vértices de X_v, concretamente los adyacentes a x. Los vértices adyacentes a x que están en X_v, debemos ahora marcarlos como cojos, ya que existe un vértice que no va a formar parte del supporting set y es adyacente a ellos (la x que hacemos forget). Y los vértices que eran adyacentes a x pero que ya hemos olvidado (no pertenecen a X_v) en su momento ya sabían que no íbamos a elegir x para la solución, por lo que ya los hemos marcado como cojos con anterioridad (concretamente al hacer su forget).

Entonces, en resumen, vemos que al hacer forget de un nodo, si el nodo no es de C (es decir que quizás si que es posible soportarlo) cabe revisar sus adyacencias con el resto de nodos de X_v. Si todas sus adyacencias dentro de X_v son de S, y el nodo también es de S, y no lo hemos “marcado como cojo anteriormente” (no pertenece a C), entonces estamos completamente seguros de que podemos sumar 1 a la puntuación de esa entrada de la tabla. En caso de que no se cumpla lo anterior, el nodo es cojo, entonces hay que sumar 0 a la puntuación (equivalente a no sumar), y en el caso de que ni siquiera esté en S, hay que añadir a C sus adyacentes en X_v (marcar como “cojos” sus adyacentes).

Finalmente también destacar que luego de todo esto, y de añadir x a Q si pertenece a S, ergo añadirlo a la solución definitiva (Q) si era parte de la solución parcial (S). Hay que eliminar x de S y de C, ya que hemos hecho forget de x y no puede aparecer más en la entrada de la tabla (pero si en la salida/solución de la tabla). Durante esta eliminación de x de la entrada, si dos entradas de la tabla pasan a tener la misma S y la misma C y la misma n, significa que debemos escoger la que tenga mejor puntuación (si su puntuación es igual y solo queremos terminar el algoritmo con una de las soluciones óptimas, podemos olvidar una de las dos). Esto se produce en un $max_p(P_1,P_2)$.

Sobre esta eliminación iterativa, sigue siendo controversial el hecho de que el algoritmo no puede coger entradas, eliminar cosas a su antojo etc… Entonces si os fijais en el forget, la solución se consigue mediante un proceso constructivo en el que miras todas las posibles entradas de antes del forget que te resultan en la misma tupla (n,S,C) tras hacer el forget. Sobre este conjunto escoges la que al hacer el forget maximiza puntuación (el infame max_p). La casuística de este conjunto es compleja y especulamos que es incompleta, es decir nos faltan casos que resultan en la misma tupla (n,S,C) tras hacer el forge, pero la vida no nos da para mas. Al menos a mi, Nico.

Join node
Aqui solo puedo intentar explicar el algoritmo como que juntamos todas las soluciones parciales posibles en todas sus combinaciones. Que quieres una tupla con 6 vertices en el suporting set? Pues te miro si es mejor juntar una solucion de 3 en el nodo hijo izquierdo del join con una solucion de 2 en el hijo derecho del join. Asi con todas las posibles n, dado que sus S y C sean compatibles. Y tras mirar todas las combinaciones, otra vez escoges la que te da mejor puntuacion.

En resumen, dados estos operadores yo estoy listo para confirmar que si sale este problema en el examen, entonces P = NP o es imposible que podamos aprobar.

Se admiten dudas de aquí para abajo……..

¿Se responderan todas las dudas?
no
¿Que es treewith?
no
¿Porque el forget es tan largo?
no

Agradecimientos especiales a Sergi que se ha currado este ejercicio