Vertex Cover in a Tree

We exploit the input structure, that is $G = (V, E)$ being a tree (connected acyclic graph) by confidently traversing *from* an arbitrary root $v \in V$. This method requires an intermediate structural translation of $G = (V, E)$ into a series of nodes with children nodes hence surfacing sub-problems, allowing for the use of this dynamic programming approach;

Algorithm 1 IncExcRootVC

Require: r, the root of the tree

for each $v \in child(r)$ **do**
    $inc[v], exc[v] \leftarrow IncExcRootVC(v)$
end for
$exc \leftarrow \bigcup_{v \in child(r)} inc[v]$
$inc \leftarrow \{r\} \cup \bigcup_{v \in child(r)} min(inc[v], exc[v])$
return inc, exc ▷ Returns $\{r\}, \emptyset$ in case of $child(r) = \emptyset$

Noting that we only really need E due to how the information of the vertices is nessecarily contained in it already (property of a tree). For any $e = (a, b) \in E$, we know $a, b \in V$. We will abritrarilly interpret this edge as having b as child of a. We just assume the edges provided are sorted, if not, we can just apply some basic lexical sorting to the edges provided. We wrap the above function in the following way;

Algorithm 2 VertexCoverOfTree

Require: $sorted(E)$, the edges

$R \leftarrow \{\}$ ▷ The set of nodes that are parentless
$C \leftarrow \{\}$ ▷ The set of nodes that have children
for each $(a, b) \in E$ **do**
    $a[child] \leftarrow a[child] \cap \{b\}$
    $R \leftarrow R \setminus \{b\}$
    $C \leftarrow C \cap \{b\}$
    if $a \notin C$ **then**
        $R \leftarrow R \cap \{a\}$
    end if
end for
$root \leftarrow R[0]$
return $min(IncExcRootVC(root))$

This algorithm works linearly, as we stitch two linear algorithms together $O(n + n) = O(n)$. Iterating once on the edges of G to construct an auxiliary structure, for which we recursively iterate down the tree and vist each vertex once. I like to think that that we traverse this tree *bottom up* even though it

doesn't quite appear that way. For leaves, we only have possible vertex covers with itself included or not at all (as per comments), we return both though for reasons that are clearer later. For internal nodes we think of the same two situations, but generalised;

- The simpler situation where it **does not** include itself, then we know it's childrens vertex covers that have the children included **must** be unioned and used in order to cover those incident edges.
- The situation where it **does** include itself, then even though natural to assume we include vertex covers of childrens that don't have it included, we're not sure from this perspective alone whether the grandchildren are covering the edges or not in the minimal vertex cover. Hence we just take the minimal vertex cover of both!

Kernelization

In order to prove safety of the proposed reduction rules we must show (ψ, k) is a yes-instance of MAX SAT **if and only if** (ψ', k') is a yes-instance of MAX SAT. Must prove both sides of implication, albeit a simple counter-example in any direction suffices to show a reduction rules ineligibility.

For reference; the boolean formula ψ is in conjunctive normal form (CNF) over variables $V = \{v_1, v_2, ..., v_n\}$ and clauses $C = \{C1, C2, ..., Cm\}$. We also know that for each $v \in V$ appears at most once in any $c \in C$.

Rule A

Lemma 1: Construction of $t : \{F, T\}$ for "unnegated" variable

Given a ψ over V and C such that if $v_i \in V$ is "unnegated" in all $c \in C$. Let $C_{v_i} \subseteq C$ be defined as all clauses mention v_i. Trivially any truth assignment $t : V \to \{F, T\}$ such that $t(v_i) = T$ satisfies all $c \in C_{v_i}$ as the clause by definition of CNF on ψ has a disjunction on v_i hence t satisfies at least $|C_{v_i}|$ clauses of C in ψ.

Lemma 1a: Such $t(v_i) = T$ definition is strictly more satisfying on C than if the same $t(v_i) = F$

Say t satisfies $C_{tF} \subseteq C$ clauses when $t(v_i) = F$. Then by the above lemma when we change $t(v_i) = T$, we satisfy all C_{v_i} clauses **also**. Hence $C_{tT} = C_{tF} \cup C_{v_i}$ and hence $|C_{tF}| \geq |C_{tT}|$.

$\Rightarrow$

Consider a yes-instance of (ψ, k). If there is some $v_i \in V$ that is "unnegated" in C, by lemma 1a, we know the truth assignment $t(v_i) = T$ is maximal relative to $t(v_i) = F$. Let $C_{\text{sat}_\psi} \subseteq C$ be all the satisfied clauses by our maximal truth

assignment on ψ. We know by lemma 1 that $|C_{\text{sat}_\psi}| = |C_{v_i}| + l \geq k$ for some l and C_{v_i} being all clauses mentioning v_i.

Building ψ' on $V' = V \setminus v_i$ and $C' = C \setminus C_{v_i}$, it's clear that we have $|C_{\text{sat}_{\psi'}}| = l \geq k - |C_{v_i}|$ showing that $(\psi', k - |C_{v_i}|)$ is a yes-instance as the rule describes.

$\Leftarrow$

Given a ψ' over V' and C'. Let $C_{\text{sat}'_\psi} \subseteq C'$ be all the satisfied clauses by some maximal truth assignment on ψ'. Suppose $|C_{\text{sat}'_\psi}| = l$.

Arbitrarily define a novel $v_i \notin V'$ and generate a series of disjunctive clauses $|C_{v_i}|$ that contain v_i "unnegated" alongside any number of $v \in V'$. Suppose $|C_{\text{sat}'_\psi}| = l \geq k - |C_{v_i}|$ for some k. Then by definition $(\psi', k - |C_{v_i}|)$ is a yes-instance. Building ψ on $V = V' \cup \{v_i\}$ and $C = C' \cup C_{v_i}$ and extending the t such that $t(v_i) = T$ then $|C_{\text{sat}_\psi}| = |C_{v_i}| + l \geq k$, hence (ψ, k) is a yes-instance.

Rule B

$\Rightarrow$

Consider a yes instance of (ψ, k) such that $C' \subseteq C$ with a maximal truth assignment $t : V \to \{T, F\}$ such that $t(x) = F$, $t(y) = T$ and $t(z) = T$. We know that all clauses in C' (defined in the rule) are satisfied. If $C \setminus C'$ has no literals x, y and z, we know that C_{sat_ψ} (as defined earlier) has a size $l + |C'| \geq k$ for some l. Define a ψ' over $V' = V \setminus \{x, y, z\}$ and $C \setminus C'$. Let $C_{\text{sat}'_\psi} \subseteq C \setminus C'$ be all the satisfied clauses the same truth assignment from ψ. Clearly $|C_{\text{sat}'_\psi}| = l \geq k - |C'| = k - 3$, hence $(\psi', k - 3)$ is a yes-instance as the rule describes.

$\Leftarrow$

Consider a yes-instance (ψ, l) acting over V' and C'. Let $C_{\text{sat}'_\psi} \subseteq C'$ be all the satisfied clauses by some maximal truth assignment on ψ'. Then $|C_{\text{sat}'_\psi}| \geq l$.

Without loss of generality arbitrarily define a novel $x, y, z \notin V'$ and construct the set of disjunctive clauses $C_{x,y,z} = \{(x \vee y), (\neg y \vee z), (\neg z \vee \neg x)\}$. Extend the truth assignment such that $t(x) = F$, $t(y) = T$ and $t(z) = T$. Then a new ψ over $V = V' \cap \{x, y, z\}, C = C' \cap C_{x,y,z}$ has clearly then $|C_{\text{sat}_\psi}| \geq l + |C_{x,y,z} = l + 3$. Hence $(\psi, l + 3)$ is a yes-instance.

Rule C

We just show a counter example; given ψ on C' and $V = \{x, y, z\}$. We know $(\psi, 3)$ is a yes-instance. For example the truth assignment $t : V \to \{T, F\}$, with $t(x) = F$, $t(y) = T$, $t(z) = T$ satisfies all clauses in C'. Applying the rule, we create ψ', which operates on the rule transformed clauses $C_{\text{rule}} = \{(y \vee y), (y \vee z), (\neg y)\}$. Note $(y \vee y) = (y)$ and there is no truth assignment

that can satisfy both y and $\neg y$. Hence $(\psi', 3)$ is a no-instance in contradiction to what the rule describes.

Depth-bounded search trees 1.

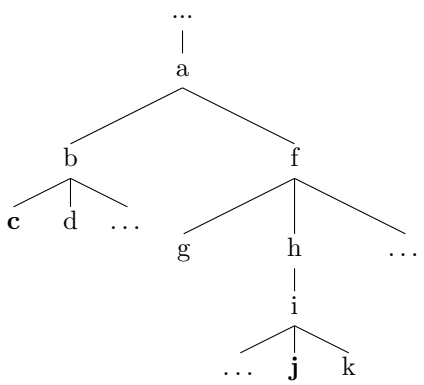
Given a graph $G = (V, E)$ and k, we construct a search tree as follows. We perform a recursive 4 path deep DFS search on each node given in V searching for 4-cycles. If no 4-cycle is found, then $G = (V, E)$ is cycle free and we return. If there is a 4-cycle and $k > 0$ (ie we can still remove vertices), choose the first 4-cycle path. It will have shape $\{(a, b), (b, c), (c, d), (d, a)\}$ for vertices $a, b, c, d \in V$. These form the branching factor of our search tree, where for each mentioned vertex, say v. We recursively step down with $G = (V \setminus \{v\}, E)$ and $k - 1$.

The running complexity of this DFS search is $O(n^4)$, whilst the search tree is bounded strictly by 4^k. Hence complexity lies in $O(4^k n^3)$. Which shows that the problem is indeed FPT.

Depth-bounded search trees 2.

Given a vertex pair in H, we can find the path between the two in a given tree T linearly. This path represents all candidate edges that can be cut to break this particular pair up. Letting a search tree depend on this path alone is susceptible to a combinatorial explosion depending on input T and hence doesn't offer a FPT algorithm no matter what the "per search" task is.

We must begin with some preparations showing that certain edges in this path are strictly "more" impactful to cut when considering other pairs in H. Given a tree $T = (V, E)$ rooted at some vertex, consider an arbitrary pair of vertices, for illustration say (c, j);



We can always find the least common ancestor (LCA_T with respect to T) uniquely.

Definition: PE_{LCA} **of** (a, b)

Let PE_{LCA} be the function that maps (a, b) for $a, b \in V$ to the set of edges on the shortest connecting $path(a, b) = \{..., (i, LCA(a, b)), (LCA(a, b), j), ...\}$ that are incident to $LCA(a, b)$.

Corollary: $1 \leq |PE_{LCA}(a, b)| \leq 2$ **when** $a \neq b$ **in** V **of a tree** $T = (V, E)$

Follows directly of the acyclic nature of a tree. There is only one shortest path between a, b. It must go through $LCA(a, b)$. Two cases to consider - one being that $LCA(a, b) \in \{a, b\}$ then clearly there is only one edge in $PE_{LCA}(a, b)$ (the terminal one) and if $LCA(a, b) \notin \{a, b\}$, then as the definition shows, there is exactly 2 edges in $LCA(a, b)$.

Definition: H **is partially ordered on** $\leq_\alpha$

Define the following reflexive and transitive partial ordering;

$$\text{for any two } (a, b), (c, d) \in H$$
$$\text{if } PE_{LCA}(a, b) \cap path(c, d) \neq \emptyset \text{ then } (a, b) \leq_\alpha (c, d)$$

Corollary $(a, b) \leq_\alpha (c, d) \in H$**,** $\exists e \in PE_{LCA}(a, b) \Rightarrow$ **forest** $G = (V, E \setminus \{e\})$ **disjoins both** (a, b) **and** (c, d)

Since $e \in PE_{LCA}(a, b) \cap path(c, d) \Leftrightarrow e \in path(a, b) \cap path(c, d)$.

Algorithm to solve TREE MULTI CUT

Similar to our first algorithm, we want $T = (V, E)$ mapped to the node/children structure of a tree (for easier sub problem splitting). Then for the recursive step we first in sequence clarify our candidate edge to remove by;

- We reduce H into M by map reducing each element (a, b) of H to pairs $(LCA(a, b), path(a, b))$, dropping elements if $LCA(a, b)$ is not found. This can be done in $|H|n$ steps. Ofcourse if $|M|$ is empty, return true with the passed k.
- In another $|H|n$ steps we map $m = (LCA(a, b), path(a, b)) \in M$ to $k = (PE_{LCA}(a, b), path(a, b)) \in K$.
- Then we reduce K with another $|H|^2 n$ to L by pairwise checking elements (a, b), (c, d) if $(a, b) \leq (c, d)$, if so drop element (c, d).

Take the first element of L, say (i, j) and then perform at **most two subsequent** calls for each $e \in PE_{LCA}(i, j)$ (by corollary, at most size 2) passing down $k - 1$ and the tree T rooted at both sides of e.

- The return of the first is either false (unable to cut), or true and the remaining cuts say l, which is
- Directly passed into the subsequent recursive call that also either returns false (unable to cut), or true and the remaining cuts say m.

We pass up true in that case alongside m.

Example run on a yes-instance such that $k = 3, |V| = 14, |H| = 5$

Consider the following yes-instance, curated to serve the best parts of this algorithm.

with $H = \{(2, 14), (3, 13), (4, 12), (5, 11), (6, 10)\}$.

Respectively we begin with the following map reductions on H

$M = \{(2, \{(2, 3), ..., (13, 14)\}),$
$(3, \{(3, 4), (4, 5), ..., (12, 13)\}),$
$(4, \{(4, 5), (5, 6), ..., (11, 12)\}),$
$(5, \{(5, 6), (6, 7), ..., (10, 11)\}),$
$(6, \{(6, 7), (8, 9), (9, 10)\})\}$

$K = \{(\{(2, 3)\}, \{(2, 3), ..., (13, 14)\}),$
$(\{(3, 4)\}, \{(3, 4), (4, 5), ..., (12, 13)\}),$
$(\{(4, 5)\}, \{(4, 5), (5, 6), ..., (11, 12)\}),$
$(\{(5, 6)\}, \{(5, 6), (6, 7), ..., (10, 11)\}),$
$(\{(6, 7)\}, \{(6, 7), (8, 9), (9, 10)\})\}$

The final reduction we end up with only one candidate in K to step into since $(6, 7)$ is an element of all other paths found.

$$\mathbf{K} = \{(\{(6, 7)\}, \{(6, 7), (8, 9), (9, 10)\})\}$$

Now the search tree expands in a bounded way to repeat the process (mind you this happens sequentially). Breaking up the graph into two by "cutting" $(6, 7)$.

With passed down
$k-1$ and original H.

6
|
5
|
4
|
3
|
...
|
1

$M = \emptyset$

Hence returning TRUE, $k-1$

With passed down
$k-1$ and original H.

6
|
7
|
8
|
9
|
...
|
14

$M = \emptyset$

Hence returning TRUE, $k-1$

Evaluation of Algorithm

This algorithm by exploration above ensures optimal progress is made in a way that selects edges that are most "impactful" (as per the $\leq_\alpha$). It the search space is bounded by 4^k, and complexity is $O(4^k(|H|n+|H|n+|H|^2n)) = O(4^k * |H|^2n)$ which shows that the problem is indeed FPT.