Parameterized Algorithms for Cycle Hitting Problems

Date: July 8, 2015

Instructor: S K Mehta

Akshay Kumar

Abstract

In this paper, we consider two different problems of computing the minimum set of vertices whose removal kills all cycles of a certain type. The first problem is Feedback Vertex Set where we want to kill all the cycles in a graph and the second problem is Even Cycle Transversal where our aim is to kill all even length cycle. While Feedback Vertex Set has proven approximation bounds, exact algorithms and parameterized algorithm, Even Cycle Transversal has not been well studied. Since both the problems are NP-complete, we parameterized exact algorithms for them. We give an $\mathcal{O}^*(5^k)$ parameterized algorithm for Feedback Vertex Set and extend it to obtain an $\mathcal{O}^*(13^k)$ parameterized algorithm for Even Cycle Transversal .

1 Introduction

Given a universe U of elements and a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of subsets of U, the hitting set problem asks for a subset $T \subseteq U$ of minimum cardinality having non-empty intersection with every subset $S_i \in \mathcal{S}$. This problem is $\mathbf{NP}-$hard [17] and admits a approximation of $\log_2 |U|$ [8] using a greedy algorithm.

Their are special instance of hitting set problem where the number of subsets, $|\mathbf{S}|$, is exponential in the size of universe, $|U|$ [6]. A greedy approximation algorithm in such a case is infeasible. Typically, $\mathbf{S}$ has a succint representation in these cases and given a set T, it can be determined, in polynomial time, whether T is a hitting set or not.

Parameterized complexity focuses on classifying problems according to their inherent difficulty with respect to multiple parameters, rather than just input size. Assuming, $\mathbf{P} \neq \mathbf{NP}$, there are many problems requiring superpolynomial time in terms of the size of the input but are polynomial time solvable in terms of input size when we consider an additional parameter k. A problem with input size n and a parameter k is said to be *fixed point tractable* (**FPT**) if there exists an algorithm to solve the problem with running time $f(k) \cdot n^{\mathcal{O}(1)}$ where f is a function of k alone. The infeasibility of greedy approximation algorithm when size of $\mathcal{S}$ is exponential in the size of U warrants a need for studying parameterized algorithm.

Given a graph, cycle hitting problems asks for a subset of vertices of the graph with non-empty intersection with every cycle of a "particular" type. We consider two different types of cycle hitting problems: Feedback Vertex Set (FVS) and Even Cycle Transversal (ECT). FVS asks for a minimum set of vertices whose deletion kills all the cycles of a graph. ECT asks for a minimum set of vertices whose deletion kills all the even length cycles of a graph. In terms of parameterized complexity, FVS (ECT) problem is: Given a graph G and a parameter k, find a subset of k vertices whose deletion kills all the cycles (even cycles, respectively) of the graph.

Feedback Vertex Set plays a prominent role in deadlock recovery in operating systems: a deadlock is represented by cycle in system resource allocation graph. To recover fromd deadlocks, we need to abort the minimum set of processes such that system resource allocation graph becomes acyclic. Feedback Vertex Set also has applications in the areas of constraint satisfaction problems and Bayesian inference [1] and in VLSI chip design.

1.1 Previous Work

The theory of parametrized complexity was developed by Downey and Fellow [12]. For a more elaborate exposition, refer the book by Flum and Grohe [14].

The basic technique we use is iterative compression, which was first discovered when designing **FPT** algorithm for Odd Cycle Transversal problem (OCT) [23]. OCT asks for a minimum subset of vertices whose deletion kills all the odd cycles of the graph, *i.e.*, makes the graph bipartite. The running time of the original **FPT** algorithm for OCT was improved to $\mathcal{O}^*(3^k)$ [20]. Recently, a linear programming based **FPT** algorithm [19] improved the running time to $\mathcal{O}^*(2.32^k)$.

FVS is one of the most well studied problem in paraterized complexity and has a long list of **FPT** algorithm for FVS includes [4, 11, 12, 22, 16, 10, 13, 7, 5, 2, 9, 18]. The best known deterministic algorithm runs in time $\mathcal{O}^*(3.6^k)$ [18]. If we allow randomization, the Cut&Count technique gives an $\mathcal{O}^*(3^k)$ algorithm [9].

While OCT and FVS has attracted a lot of attention, ECT har largely been left ignored. Kakimura et. al. [15] looked at a generalization of ECT, Subset Even Cycle Transversal. In this problem, apart from the graph G, an additional vertex set $T \subseteq V(G)$ is given as input. The objective is to find a minimum vertex set $S \subseteq V(G)$ such that $G - S$ does not contain any even cycle with non-empty intersection with T. $T = V(G)$ corresponds to Even Cycle Transversal problem. The algorithm described in the paper uses graph minor machinery and has running time $2^{2^{2^k}}$. Misra et. al. [21] give an $\mathcal{O}^*(50^k)$ algorithm for Even Cycle Transversal . They used iterative compression and give a branching algorithm with branching factor 7.

The main contributions of this paper are two fold: a) We present an $\mathcal{O}^*(5^k)$ algorithm for Feedback Vertex Set which simplifies the analysis of the algorithm presented by Chen et. al. [7]. b) We present an $\mathcal{O}^*(13^k)$ algorithm for Even Cycle Transversal which beats the currently best known bound of $\mathcal{O}^*(50^k)$ presented by Misra et. al. [21].

2 Preliminaries

Notation. For a graph G, we denote the vertex set of G by $V(G)$ and edge set of G by $E(G)$. Alternately, the notation $G(V, E)$ or $G = (V, E)$ is also used. Let $S \subseteq V(G)$. By $G[S]$, we mean the subgraph induced on S. The subgraph $G[V(G) \setminus S]$ is denoted by $G - S$. If G' is a subgraph of G, $V(G')$ denotes the vertex set of G' and $G - G'$ denotes the subgraph $G[V(G) - V(G')]$. $G' \cup S$ denotes the subgraph $G[V(G') \cup S]$. Similarly, if G_1 and G_2 are two subgraphs of G, $G_1 \cup G_2$ denotes the subgraph $G[V(G_1) \cup V(G_2)]$. If P and P' are two paths, $P' \subseteq P$ denotes the fact that P' is a subpath of P. Set $\{1, 2, \ldots, n\}$ is denoted by $[n]$.

Definition 1 *A set of vertices is called a fvs (feedback vertex set) if their removal kills all the cycles in the graph.*

Definition 2 *A set of vertices is called an ect (even cycle terminator) if their removal kills all the even cycles in the graph.*

Problem 1 Feedback Vertex Set ***(FVS)*** *Given an undirected unweighted graph G and a positive integer k, does there exist a fvs of G of size k? If yes, find it.*

Problem 2 Even Cycle Transversal ***(ECT)*** *Given an undirected unweighted graph G and a positive integer k, does there exist a ect of G of size k? If yes, find it.*

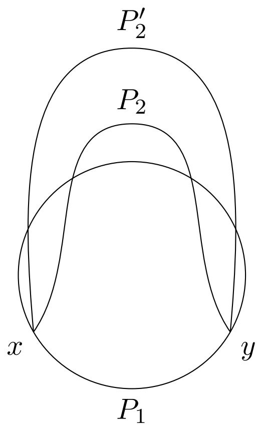

Figure 1: $C_2 = P_2 \cup P_2'$.

Observation 3 *Let G be a graph and S be a fvs of G. Then $G - S$ is a forest.*

Definition 4 *An odd graph is a graph without any even cycles.*

Definition 5 *A cactus graph is a connected graph in which two cycles share at most one vertex in common. An odd cactus graph is a graph which is an odd graph as well as a cactus graph.*

For simplicity, we will slightly abuse the notation and say that a graph is a cactus graph if each of its connected component is a cactus graph.

A biconnected graph is a graph with vertex connectivity two, *i.e.*, removal of any one vertex still leaves the graph connected. In other words, a biconnected graph is a graph with no articulation points. A biconnected component, equivalently a block, of a graph is a maximally biconnected subgraph. Hence, every block of a cactus is either an isolated vertex (if a connected component is simply an isolated vertex), K_2 or a cycle.

Lemma 6 *If there are two cycles C_1 and C_2 in a graph G which share two or more vertices in common, there's an even cycle in G.*

Proof: If C_1 or C_2 is even, there's nothing to prove. Hence, assume that both C_1 and C_2 are odd cycles. Let S be the set of vertices which are common in C_1 and C_2 such that $|S| \geq 2$. Let $x, y \in S$ be such that one of the paths connecting x to y in C_1, say P_1, contains no other vertex belonging to S. Let P_2 and P_2' be the two paths from x to y in C_2. Since C_2 is an odd cycle, P_2 and P_2' have different parities. Therefore, cycles $P_1 \cup P_2$ and $P_1 \cup P_2'$ have different parities. Hence, one of the cycles must be of even length. See Figure 1 for an illustration.

□

Corollary 7 *Let G be a graph and S be an ect of G. Then $G - S$ is an odd cactus graph.*

Hardness of Feedback Vertex Set . Hardness of FEEDBACK VERTEX SET has been established by reducing VERTEX COVER problem to FVS [17]. We present this result for the sake of completeness.

Definition 8 *The vertex cover of a graph is a set of vertices such that every edge of the graph is incident on at least one vertex of the set.*

Definition 9 Vertex Cover *Problem* *Given a graph G and a positive integer k, find a vertex cover of G of size k, if it exists.*

We reduce a Vertex Cover instance (G, k) to a Feedback Vertex Set instance (H, k) as follows:

$$V(H) = V(G) \cup \{h_{uv} \mid (u, v) \in E(G)\}$$
$$E(G) = \{(u, h_{uv}), (h_{uv}, v), (u, v) \mid (u, v) \in E(G)\}$$

The idea is to "convert" every edge into a triangle. Covering of an edge corresponds to killing that triangle and vice versa.

Observation 10 *A vertex of the form h_{uv} of the second type has exactly two neighbors: u and v.*

Lemma 11 *G has a vertex cover of size k iff H has a fvs of size k.*

Proof: We will show that any $k-$sized vertex cover of G is a *fvs* of H as well. Consider a cycle in H involving some vertex of type h_{uv}. By the above observation, this cycle must also contain the vertices u and v, which forms an edge in G. This means at least one of u or v must be in the vertex cover, which kills the cycle. For cycles in H involving no vertex of the type h_{uv}, all the edges are killed because each edge must be covered by the vertex cover.

If we have a $k-$sized *fvs* F of H, we can construct a vertex cover F' of G of size at most k. To construct F', replace each vertex of the type h_{uv} by u (or, v) and eliminate duplicates. Corresponding to every edge $(u, v) \in E(G)$, there exists a cycle $(u, h_{u,v}, v)$ in H. Hence one of u, v or $h_{u,v}$ must be in F. This implies either u or v is present in F'. Hence, at least one end point of every edge of G is present in F'.

□

Hardness of Even Cycle Transversal . Hardness of Feedback Vertex Set can be established by a reduction from Feedback Vertex Set problem. At the top level, the reduction involves replacing every cycle by an even cycle.

We reduce a Feedback Vertex Set instance (G, k) to an Even Cycle Transversal instance (H, k) as follows:

$$V(H) = V(G) \cup \{h_{uv} \mid (u, v) \in E(G)\}$$
$$E(H) = \{(u, h_{uv}), (h_{uv}, v) \mid (u, v) \in E(G)\}$$

Observation 12 *Vertices of the form $h_{u,v}$ in H have exactly two neighbors: u and v.*

The idea is to subdivide each edge (u, v) into (u, h_{uv}) and (h_{uv}, v). Any cycle $v_1, v_2, \ldots, v_t$ in G is replaced by a cycle $v_1, h_{v_1,v_2}, v_2, h_{v_2,v_3}, \ldots, h_{v_{t-1},v_t}, v_t, h_{v_t,v_1}$ in H, *i.e.*, a cycle of length t gets converted into a cycle of length $2t$. Conversely, every cycle in H is of the form $v_1 h_{v_1 v_2} v_2 h_{v_2 v_3} v_3 \ldots h_{v_{t-1} v_t} v_t h_{v_t v_1}$ where $v_1 v_2 \ldots v_t$ is a cycle in G.

Observation 13 *There is a one to one correspondence between the cycles in G and that of H. Under this correspondency, a cycle of length t in G corresponds to a cycle of length $2t$ in H.*

Lemma 14 *G has a fvs of size k iff H has a ect of size k.*

Proof: Observe that if $S \subseteq V(G)$ is a *fvs* of G, then S is trivially an *ect* of G. To prove the other direction, let S be an *ect* of H. Replace each vertex of type h_{uv} in S by u (*i.e*, one of u or v). Let the resulting set be S'. Then S' is also a *fvs* of G. Observe that $|S'| \leq |S|$.

□

We give an $\mathcal{O}^*(5^k)$ algorithm for Feedback Vertex Set and extend it to obtain an $\mathcal{O}^*(13^k)$ algorithm for Even Cycle Transversal . This improves the currently best known bound of $\mathcal{O}^*(50^k)$ for Even Cycle Transversal [21].

To solve the two problems, we will look at their iterative compression version:

Problem 3 ***Iterative compressive*** **Feedback Vertex Set** ***(IC-FVS)*** *Given a graph G and a $(k+1)-$sized fvs S of G, find a $k-$sized fvs S', if it exists.*

Problem 4 ***Iterative compressive*** **Even Cycle Transversal** ***(IC-ECT)*** *Given a graph G and a $(k+1)-$sized ect S of G, find a $k-$sized ect S', if it exists.*

Theorem 15 *An FPT algorithm for IC-FVS (IC-ECT, respectively) gives an FPT algorithm for FVS (ECT, respectively).*

Proof: Let $V(G) = \{v_1, v_2, \ldots, v_n\}$. Define $G_i := G[\{v_1, v_2, \ldots, v_i\}]$. Suppose G_i has a $k-$sized *fvs* F_i. Then G_{i+1} has a $(k+1)-$sized *fvs* , $F_i \cup \{v_{k+1}\}$. Using IC-FVS algorithm, compute a $k-$sized *fvs* F_{i+1} for G_{i+1}. Repeat this till a $k-$sized *fvs* for G is computed. Observe that $f + K := \{v_1, v_2, \ldots, v_k\}$ is obviously an *fvs* of G_K, which is the base case for this iterative procedure.

Moreover, if the time complexity of IC-FVS is $\mathcal{O}(f)$, the time complexity of FVS is $\mathcal{O}(n \cdot f)$. Same argument holds for *ect* as well.

Since k is not known apriori, we can guess a value of k and run this algorithm. If it succeeds, then make the next guess lower else higher. Using binary search, we can compute an *fvs* (*ect*) of optimum size in $\mathcal{O}(n \cdot \log n \cdot f)$ time. □

We will try to devise a solution for IC-FVS (IC-ECT) problem. Given a graph G and a $(k+1)-$sized *fvs* S, let S' be a $k-$sized *fvs* of G. Let $Y = S \cap S'$, $N = S \setminus S'$ and $N' = S' \setminus S$ (See Figure 2). Towards constructing S', fix Y. In the induced subgraph $G - Y$, we intend to find a *fvs* N' of size at most $k - |Y|$, which is disjoint from N. We will incrementally construct N'. This process is repeated for every possible subset Y of S till a $k-$sized *fvs* (*ect* , respectively) is found.

Since S is a *fvs* , by Corollary 3, $G - S$ is a forest. For S' to be a *fvs* , N must also be a forest. Similarly, for *ect* , we can say that both $G - S$ and N must be odd cactus.

Hence, we define the following variant of both the problems, where $H = G - Y$:

Problem 5 ***Disjoint*** **Feedback Vertex Set** ***(DISJOINT-FVS)*** *Given a graph H and a subset $N \subseteq V(H)$ such that both $H[N]$ and $H - N$ are forests, find a subset of vertices $N' \subseteq V(H - N)$ of size at most k' such that N' is a fvs of H. We denote its instance by (H, N, k').*

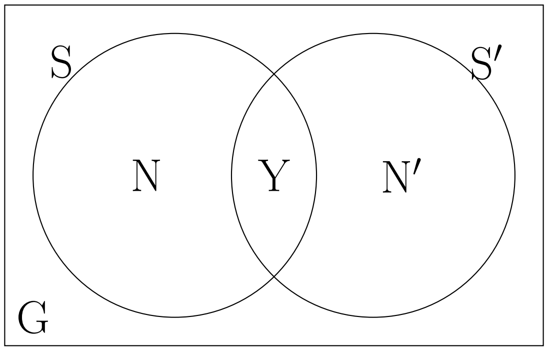

Figure 2: Partitioning of S into N and Y. The new *fvs* (*ect*) S' is composed of Y and some vertices from $G - S$ but no vertex from N.

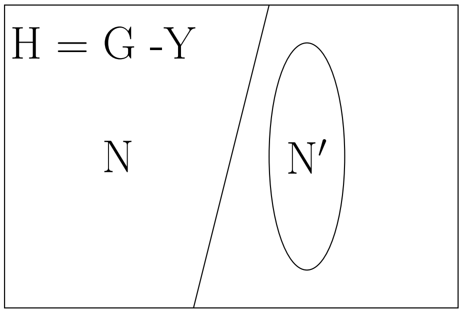

Figure 3: Disjoint version of FVS and ECT. Both $H[N]$ and $H - N$ are forest (odd cactus). We need to find a subset of $N' \subseteq H - N$ of a specified size k' such that $H - N'$ is a forest (odd cactus, respectively).

Problem 6 ***Disjoint*** **Even Cycle Transversal** ***(DISJOINT-ECT)*** *Given a graph H and a subset $N \subseteq V(H)$ such that both $H[N]$ and $H - N$ are odd cactus, find a subset of vertices $N' \subseteq V(H - N)$ of size at most k' such that N' is a ect of N. We denote its instance by (H, N, k').*

See Figure 3 for an illustration.

Theorem 16 *An FPT algorithm for DISJOINT-FVS (DISJOINT-ECT) gives an FPT algorithm for IC-FVS (IC-ECT, respectively). Moreover, if the running time of former algorithm is $\mathcal{O}^*(c^\alpha)$ (for some constant c), the running time of the algorithm for iterative compressive version will be $\mathcal{O}^*((c + 1)^\alpha)$ where $\alpha \leq k$.*

Proof: Let (G, S) be an instance of IC-FVS such that $|S| = k + 1$. We branch into at most $2^{|S|} = 2^{k+1}$ subcases, guessing the intersection of the solution S' with the set S. Let the current guess be $Y \subseteq S$. We remove Y from G and call the DISJOINT-FVS instance $(H := G - Y, N := S - Y, k - |Y|)$. Iterating over all possible subsets Y of S, the total running time is

$$\mathcal{O}^*\left(\sum_{Y \subseteq S} c^{k-|Y|}\right) = \mathcal{O}^*\left(\sum_i \sum_{\substack{Y \subseteq S \\ |Y|=i}} c^{k-|Y|}\right) = \mathcal{O}^*\left(\sum_i \binom{k}{i} c^{k-i}\right) = \mathcal{O}^*\left(\sum_i \binom{k}{i} c^i\right) = \mathcal{O}^*((c+1)^k)$$

The same can be argued about IC-ECT as well. □

3 Feedback Vertex Set

This section establishes Theorem 17.

Theorem 17 FEEDBACK VERTEX SET *can be solved in* $\mathcal{O}^*(5^k)$ *time.*

By Theorem 15 and Theorem 16, to prove Theorem 17, it suffices to solve DISJOINT-FVS. Before moving further, we consider two different problems.

Problem 7 Lowest Common Ancestor *Given a rooted tree* T *and a set of pair of nodes* $\{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$*, find the lowest common ancestor of each pair of nodes* (s_i, t_i) $\forall\ i \in [k]$.

Theorem 18 (cf.: [3]) *There exists a linear time algorithm for solving* LOWEST COMMON ANCESTOR *problem.*

Denote the lowest common ancestor of u and v by $LCA(u, v)$.

Problem 8 Multicut on Trees *Given a tree* T *and a set of pairs of nodes* $\{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$*, find a minimum set* S *whose removal disconnects* (s_i, t_i) $\forall\ i \in [k]$*. The nodes* $s_1, t_1, s_2, t_2, \ldots, s_k, t_k$ *are called terminal nodes.*

Deterministic polynomial time algorithm for Multicut on Trees . Multicut problem for general graphs is a hard problem. Fortunately, there exists a polynomial time algorithm for the multicut problem on trees which uses the algorithm of Problem 7 as a subroutine.

Root the tree arbitrarily. We then compute LCA of each pair of terminal node using the algorithm of Theorem 18. Let $LCA(s_{i_0}, t_{i_0})$ be the node with the maximum depth out of all $LCA(s_i, t_i)$.

Lemma 19 *There exists an optimal solution* S *to multicut problem on trees such that* $LCA(s_{i_0}, t_{i_0}) \in S$.

Proof: Since the pair (s_{i_0}, t_{i_0}) is to be disconnected, one vertex from the unique path joining s_{i_0} and t_{i_0} must be present in S. Suppose a vertex v, not equal to $LCA(s_{i_0}, t_{i_0})$, is picked from this unique path. Since $LCA(s_{i_0}, t_{i_0})$ has maximum depth among all LCA nodes, if v disconnects some other pair of node $(s_{i'}, t_{i'})$, the path joining $s_{i'}$ and $t_{i'}$ must also pass through $LCA(s_{i_0}, t_{i_0})$. Hence, v can be replaced by $LCA(s_{i_0}, t_{i_0})$ in the solution set. □

Lemma 19 gives a polynomial time algorithm for multicut problem on trees: Maintain a set of pairs of terminal nodes which are to be disconnected and compute their LCAs. Add the LCA with maximum depth to the solution set and delete the pairs of terminal nodes which get disconnected. Repeat this process till all pairs of terminal nodes are disconnected. The overall Algorithm is described as Algorithm 1.

We will now focus on the problem DISJOINT-FVS. The particular instance of DISJOINT-FVS we consider is: Given a graph H and a subset $N \subseteq V(H)$ such that both $H[N]$ and $H - N$ are forests, compute a subset of vertices of $N' \subseteq V(H - N)$ of size at most k' which is a *fvs* of H.

Definition 20 *A connected component* T *of* $H - N$ *is called a branching tree if* $T \cup N$ *contains a cycle or* T *has adjacencies to more than one connected component of* N*. Therefore, if* T *is a branching tree ,* $T \cup N$ *either has a cycle or has fewer connected components than* N.

Algorithm 1 Multicut on Trees

```
1: function MulticutTrees($T, ter := \{(s_1,t_1),(s_2,t_2),\dots,(s_k,t_k)\}$)
2:     if $ter$ is empty then
3:         Return $\varnothing$
4:     else
5:         for all $i \in [k]$ do
6:             $l_i \leftarrow LCA(s_i,t_i)$
7:         end for
8:         $l_{i_0} \leftarrow$ Node with the maximum depth among $l_i$'s
9:         $ter' \leftarrow \{(s_i,t_i) \mid l_{i_0}$ does not lie on the path joining $s_i$ to $t_i\}$
10:        Return $l_{i_0} \cup$ MulticutTrees($T, ter'$)
11:    end if
12: end function
```

We use branching trees to compute N'. For each connected component T of $H - N$, either by deleting vertices from T or moving a subtree of T to N, we ensure that finally, $T \cup N$ is acyclic and edges from T are incident in at most one connected component of N.

Let $P_T(x,y)$ denote the unique path joining nodes x and y in the tree T.

Definition 21 *A pair of nodes (s,t) of a tree T is called a branching terminal pair if $H[P_T(s,t) \cup N]$ is contains a cycle or $P_T(s,t)$ has adjacencies to more than one connected component of $H[N]$.*

Observation 22 *The set of branching terminals in a branching tree T can be computed in polynomial time.*

Observation 22 follows from the fact that given a pair of node (s,t) of T, it can be checked whether (s,t) is a branching terminal pair (in polynomial time) by checking the branching terminal pair conditions for (s,t).

Denote the set of branching terminal pair of a branching tree T by ter. Root T arbitrarily. We compute the LCA of each pair of nodes contained in ter. Let $(s_0,t_0) \in ter$ be the pair of node whose LCA has maximum depth among all LCAs. In case of ties, we pick one arbitrarily. If we intend to disconnect all pairs of nodes of ter, by Lemma 19, there exists an optimal solution which contains $LCA(s,t)$.

- If $P_T(s_0,t_0) \cup N$ contains a cycle, (s,t) must be disconnected. In this case, move $LCA(s_0,t_0)$ to the solution set and delete it from the graph.
- Otherwise, $P_T(s_0,t_0) \cup N$ has fewer connected components than N. In this case, we move the subtree rooted at $LCA(s,t)$ to N.

The justification of the second step is provided when we discuss the correctness of the algorithm. We call (s_0,t_0) as the maximum branching terminal pair of T.

The overall algorithm is: If there exists a branching tree T, root it arbitrarily. Let (s_0,t_0) be the maximum branching terminal pair of T. If $P_T(s_0,t_0) \cup N$ has a cycle, delete $LCA(s_0,t_0)$ from the graph and add it to the solution set. Otherwise, we branch. The first branch is analogous to the previous case: we delete $LCA(s_0,t_0)$ from the graph and move it to the solution set. In the second branch, we move the subtree rooted at $LCA(s_0,t_0)$ from $G - N$ to N.

The entire algorithm is presented as Algorithm 2.

Algorithm 2 Parametrized Algorithm for Feedback Vertex Set

```
1:  function FeedbackVertexSet($(G, k)$)
2:      $G' \leftarrow$ Graph induced over $k+1$ randomly selected vertices of $G$
3:      $S \leftarrow V(G')$
4:      while $V(G') \neq V(G)$ do
5:          if IC-FVS($G', S$) exists then
6:              $S' \leftarrow$ IC-FVS($G', S$)                              ▷ $|S'| = k$
7:              Add a new vertex $v$ to $G'$
8:              $S \leftarrow S' \cup \{v\}$                                  ▷ $|S| = k+1$
9:          else
10:             exit the for loop
11:         end if
12:     end while
13:     if $V(G) = V(G')$ and IC-FVS($G', S$) exists then
14:         return IC-FVS($G, S$)
15:     else
16:         return no solution exists
17:     end if
18: end function

19: function IC-FVS($(G, S)$)
20:     $k \leftarrow |S| - 1$          ▷ Size of new fvs $S'$ should be one less than that of $S$
21:     for all subsets $Y$ of $S$ do
22:         $N \leftarrow S - Y$
23:         if DISJOINT-FVS($G - Y, N, k - |Y|$) exists then
24:             $S' \leftarrow$ DISJOINT-FVS($G - Y, N, k - |Y|$)
25:             exit the for loop
26:         else
27:             continue
28:         end if
29:     end for
30:     if no $S'$ is found then
31:         return no solution exists
32:     else
33:         return $S'$
34:     end if
35: end function
```

```
36: function DISJOINT-FVS($H, N, k'$)
37:     if (Both $N$ and $H - N$ are not forests) or ($k' < 0$) then
38:         return no solution exists
39:     end if
40:     if $k' = 0$ then
41:         if $H$ is a forest then return $\varnothing$
42:         else return no solution exists
43:         end if
44:     end if
45:     if there exists a connected component $T$ of $H - N$ $s.t.$ $T \cup N$ contains a cycle or has fewer
    connected components than $N$ then
46:         $(s_0, t_0) \leftarrow$ maximum branching terminal pair of $T$            ▷ Computed from Lemma 19
47:         $T' \leftarrow$ tree rooted at $LCA(s_0, t_0)$
48:         if $P_T(s_0, t_0) \cup N$ contains a cycle then
49:             if DISJOINT-FVS($H \setminus \{LCA(s_0, t_0)\}, N, k' - 1$) exists then
50:                 $S_1 \leftarrow$ DISJOINT-FVS($H \setminus \{LCA(s_0, t_0)\}, N, k' - 1) \cup \{LCA(s_0, t_0)\}$
51:             end if
52:         end if
53:         if $P_T(s_0, t_0) \cup N$ has fewer connected components than $N$ then
54:             if DISJOINT-FVS($H \setminus \{LCA(s_0, t_0)\}, N, k' - 1$) exists then
55:                 $S_2 \leftarrow$ DISJOINT-FVS($H \setminus \{LCA(s_0, t_0)\}, N, k' - 1) \cup \{LCA(s_0, t_0)\}$
56:             end if
57:             if DISJOINT-FVS($H, N \cup T', k$) exists then
58:                 $S_3 \leftarrow$ DISJOINT-FVS($H, N \cup T', k$)
59:             end if
60:         end if
61:         if $P_T(s_0, t_0) \cup N$ contains a cycle then
62:             if $S_1$ has been defined previously then return $S_1$
63:             else return no solution exists
64:             end if
65:         else
66:             if either $S_2$ or $S_3$ was been defined previously then return it
67:             else return no solution exists
68:             end if
69:         end if
70:     else
71:         return $\varnothing$
72:     end if
73: end function
```

Time complexity of the algorithm DISJOINT(H, N, k'). To compute the time complexity of the above algorithm, we will look at the following parameter:

$$\mu = k' + \text{ number of connected components of } N \tag{1}$$

When we delete a vertex and move it to the solution set, k' decreases by one. When the subtree rooted at a LCA node it moved to N, number of connected components of N decreases by one. In either case, μ decreases by one. Initially, $\mu \leq k + |N|$. Also, initially, $|N| \leq k+1$ and $k' \leq k$. Also, the IC-FVS instance always deletes a subset Y before calling DISJOINT-FVS. Hence, $|N| \leq k$. This implies $\mu \leq 2k+1$. Since the branching factor of this algorithm is two, overall time complexity is $\mathcal{O}^*(2^\mu) = \mathcal{O}^*(4^k)$. By Theorem 16, the time complexity of IC-FVS is $\mathcal{O}^*(5^k)$. Hence, FVS has a time complexity of $\mathcal{O}^*(5^k)$.

Proof of correctness of the algorithm. For a branching tree T, let (s_0, t_0) be the maximum branching terminal pair of T. If $P_T(s_0, t_0) \cup N$ contains a cycle, from Lemma 19, we know that there exists an optimal solution containing $LCA(s_0, t_o)$. In other words, (H, N, k') has the same solution has $\{LCA(s_o, t_0)\} \cup (H \setminus \{LCA(s_0, t_0)\}, N, k' - 1)$.

If $P_T(s_0, t_0) \cup N$ does not contain a cycle but has fewer connected components than N, we can't say for sure if $LCA(s_0, t_0)$ is to be added to the solution set or not. If there's a cycle C spanning multiple connected components of $H - N$ such that $V(C) \cap V(T) = P_T(s_0, t_0)$, then picking $LCA(s_0, t_0)$ might not be optimal since C can be also killed by picking vertice(s) from some other connected component of N. Hence, in addition to deleting $LCA(s_0, t_o)$ and moving it to the solution set, we branch and move the subtree T' rooted at $LCA(s_0, t_o)$ from $H - N$ to N, *i.e.*, the solution of (H, N, k') is same as the solution of $(H, N \cup T', k)$. Moving T' to N is reasonable because this step has zero cost associated with it, *i.e.*, no vertex added to the solution set.

We now argue that the presented algorithm always terminates. When we add a vertex to the solution set, k' decreases by one. When we move a subtree from $H - N$ to N, number of connected component of N decreases by one. In either case, μ decreases by one.

Note that we could have simply branched without moving subtree rooted at $LCA(s, t)$ to N but this would not have guaranteed the termination of the algorithm.

We now discuss the base case of this algorithm. Note that $\mu \geq 1$. The base case occurs when we call the instance $(H, N, 0)$. If H is a forest, then we return empty set as the solution. Else, no solution exists. Also, if at any point during the execution of the algorithm, H becomes a forest, no more vertices need to be added to the *fvs* and we simply return empty set as the solution.

4 ECT

Currently, the best known algorithm for Even Cycle Transversal problem [21] runs in $\mathcal{O}^*(50^k)$ time. Their solution relies on finding a set of vertices six vertices such that there exists an optimal solution having at least one vertex from this set. We refine their approach and crucially scrutinize the graph to arrive at a smaller such set of vertices. In this way, we bring down the time complexity to $\mathcal{O}^*(10^k)$.

Theorem 23 Even Cycle Transversal *can be solved in* $\mathcal{O}^*(10^k)$ *time.*

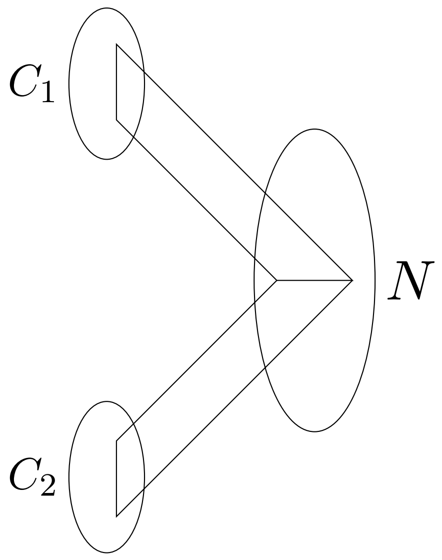

Figure 4: Two odd cycles whose union forms an even cycle in $C_1 \cup C_2 \cup N$

To prove Theorem 23, we will present an $\mathcal{O}^*(9^k)$ algorithm for DISJOINT-ECT. By Theorem 16 and Theorem 15, the time complexity for ECT would be $\mathcal{O}^*(10^k)$.

We will now focus on the following instance of DISJOINT-ECT: Given a graph H and a subset $N \subseteq V(H)$ such that both $H[N]$ and $H - N$ are odd cactus, compute a subset of vertices $N' \subseteq V(H - N)$ of size at most k' which is an *ect* of H.

We work along the lines of the algorithm described for DISJOINT-FVS and update $H - N$ and $H[N]$ such that, each connected component C of $H - N$ is such that: *a*) there is no even cycle in the graph $C \cup N$; and *b*) all edges from C are incident in the same connected component of $H[N]$.

The best way to visualize C is like a tree of odd cycles as shown in Figure 12.

However, unlike the algorithm for DISJOINT-FVS, there might still be even cycles remaining: let C_1 and C_2 be two connected components of $H - N$ such that $C_1 \cup N$ contains an odd cycle and $C_2 \cup N$ also contains an odd cycle. If the two odd cycles share more than one vertex in common, by Lemma 6, there is an even cycle in the induced graph $C_1 \cup C_2 \cup N$. See Figure 4 for an illustration. We deal with such cycles using the structure provided after ensuring conditions mentioned in the previous paragraph.

We call the first part of the algorithm as First Phase and second part, which deletes the even cycles running across more than one connected component of $H - N$, as Second Phase.

4.1 First Phase

Definition 24 *A vertex of $H - N$ is called a port vertex if it has neighbors in N. A non-port vertex is a vertex which is not a port vertex.*

Let C be a connected component of $H - N$. Note that C is an odd cactus.

Definition 25 *A path P in C is called a branching path if $P \cup N$ has an even cycle in which P is a subpath or P has adjacencies to more than one connected components of N. It is possible that P is of length zero (*i.e.*, P may be a single vertex).*

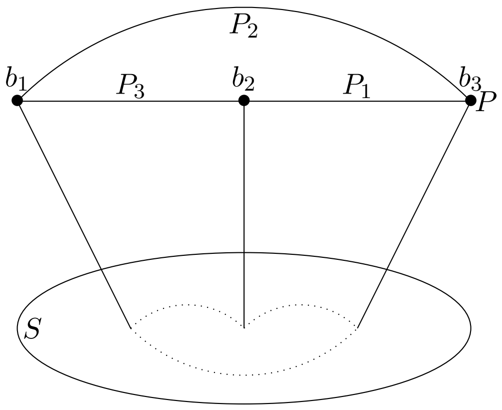

Figure 5: Branching path in a odd cactus having at least three port vertices

Remark 26 *If P is a branching path, then either $P \cup N$ has even cycle(s) or $P \cup N$ has fewer connected components than in N.*

Lemma 27 *Let C be a connected component of $H - N$ such that there are at least three edges going from C to N. Then, there exists a branching path P in C. Moreover, P can be computed in polynomial time.*

Proof: Let the three edges going from C to N be incident on port vertices b_1, b_2 and b_3 in C. Suppose the path joining b_1 to b_2 in C is P_3, b_2 to b_3 in C is P_1 and b_3 to b_1 in C is P_2. If any two of edges going from b_1, b_2 or b_3 are incident in different connected components of N, then the path P_i between those vertices is a branching path. Else, it must be the case that all edges are incident in the same connected component of N. If both P_3 and P_1 are not branching paths, $P_3 \cup N$ and $P_1 \cup N$ must have only odd cycles. Since the odd cycles in $P_3 \cup N$ and $P_1 \cup N$ share two vertices in common (b_2 and the neighbor of b_2 in N), by Lemma 6, there exists an even cycle in $P_1 \cup P_3 \cup N$. □

Corollary 28 *Let P be a path with three port vertices. Then there exists a subpath $P' \subseteq P$ such that P' is a branching path.*

Lemma 29 *Given an undirected unweighted graph H and two vertices x and y of H, the parities of all paths between x and y can be determined in polynomial time.*

Proof: Construct a new graph H' as follows:

$$V(H') = \{(v, \mathsf{even}), (v, \mathsf{odd}) \mid v \in V(H)\}$$
$$E(H') = \{((u, \mathsf{even}), (v, \mathsf{odd})), ((u, \mathsf{odd}), (v, \mathsf{even})) \mid (u, v) \in E(H)\}$$

The idea is to keep track of the parity of path encountered. An even path between two vertices u and v in H exists iff a path between (u, even) and (v, even) (or, (u, odd) and (v, odd)) in H' exists. Similarly, an odd path between two vertices u and v in H exists iff a path between (u, even) and (v, odd) (or, (u, odd) and (v, even)) in H' exists. □

Lemma 30 *Let x and y be two vertices of an odd cactus C. It is possible to find, it it exists, a path betweeen x and y of a given parity, say par, in polynomial time.*

Proof: Let P be the shortest path between x and y in C. If parity of P is *par*, P is the required path. Else, if there exists a path of parity *par* between x and y, P must pass through an odd cycle, say c, in C. c can be found by considering the block decomposition of C. By switching the vertices of c in P by the other set of vertices of c, we get a path of parity *par*. If there does not exist any such odd cycle c, there does not exist a path of parity *par* between x and y in C.

□

Note that Lemma 29 talks about existence of a path of a given parity between two vertices in a graph. Lemma 30 actually constructs a path of a given parity between two vertices in an odd cactus.

Lemma 31 *Let C be a connected component of $H - N$ such that for some connected component N' of N, $C \cup N'$ contains an even cycle or $C \cup N$ has fewer connected components than in N. Then there exists a path P such that for some connected component N' of N, $N' \cup P$ has an even cycle in which P is a subpath. Otherwise for any path P' in C and any connected component N' of N, $P' \cup N'$ does not contain an even cycle in which P' is a subpath, but there exists a path P such that $N \cup P$ has fewer connected components than in N. Further, P can be computed in polynomial time.*

Proof: For each pair of edges (b_1, c_1) and (b_2, c_2) where $b_1, b_2 \in V(C)$ and $c_1, c_2 \in V(N)$, if there exists a path P between b_1 and b_2 and a path Q between c_1 and c_2 such that P and Q have the same parity, then P is the desired path.

If no path is found as described above, then select two edges (b_1, c_1) and (b_2, c_2) such that $b_1, b_2 \in V(C)$; and c_1 and c_2 belong two distinct connected components of N. Then any path P between b_1 and b_2 is the required path. □

Lemma 32 *Let x and y be two vertices of a connected component C of $H - N$. Then, a branching path P connecting x to y in C can be found in polynomial time, if it exists.*

Proof: Let P be a branching path with endpoints x and y. Let P' and P'' be the parts of P in C and N respectively. Since P is of even length, P' and P'' have same parities.

To find P, for all vertices x' and y' of N which are neighbor of x and y respectively, check if there exists a path connecting x to y in C and a path connecting x' to y' in N having same parities. By Lemma 29, the parities of all the paths connecting x to y in C and x' to y' in N can be computed in polynomial time. We can then use Lemma 30 to contruct a path of the required parity. □

Definition 33 *A simple cactus is a cactus where no vertex belongs to more than one cycle. An odd simple cactus is a simple cactus which is also an odd cactus.*

Note that any simple cactus can be viewed as a tree each of whose nodes are either odd cycles or a vertex of the orignal simple cactus that does not belong to any cycle. We first give an algorithm for DISJOINT-ECT problem for the case when $H - N$ is a simple cactus. We will then extend the solution to the general case.

Tree representation of a simple cactus. We describe the algorithm for converting a simple cactus C into a tree T. Let the set of cycles in C be $Cyc = \{c_1, c_2, \ldots, c_m\}$ (can be found in polynomial time by considering the block decomposition of C [24]). Note that since C is a simple cactus, each block is either a cycle or K_2; and all cycles are vertex disjoint.

The nodes of T are:

$$V(T) = \{c \mid c \in Cyc\} \cup \{u \mid u \text{ is not part of any cycle in } C\}$$

The tree T is the result of contracting each cycle c into a node c. We call the nodes of first type as cycle nodes and the nodes of second type as vertex nodes. Let T_C denote the set of cycle nodes and T_V denote the set of vertex nodes. For a node $u \in V(T)$, the notation $V(u)$ denotes the set of vertices of C present in u, *i.e.*, $V(u) = V(u)$ if u is a cycle in C and $\{u\}$ otherwise. For completeness, we define the edges of the tree as well:

$$\begin{aligned} E(T) =& \{(n_1, n_2) \mid u, v \in T_V, (u, v) \in E(C)\} \ \cup \\ & \{(n_1, n_2) \mid n_1 \in T_C, n_2 \in T_V, \exists\ u \in V(n_1)\ s.t.\ (u, n_2) \in E(C)\} \ \cup \\ & \{(n_1, n_2) \mid n_1, n_2 \in T_C, \exists\ u \in V(n_1)\ \exists\ v \in V(n_2)\ s.t.\ (u, v) \in E(C)\} \end{aligned}$$

Let r be the root node of T chosen arbitrarily. Additionally, we introduce a function $\lambda \colon V(T) \to E(C)$ which maps the edges from $E(T)$ to the corresponding edge in $E(C)$. Let v be the parent of a node u. $\lambda(u)$ is defined as follows:

$$\lambda(u) = \begin{cases} (u, v) & \text{if } \ u \in T_V \text{ and } v \in T_V \\ (u, x) & \text{if } \ u \in T_V \ , \ v \in T_C \text{ and } \exists\ x \in V(v)\ s.t.\ (u, x) \in E(C) \\ (x, v) & \text{if } \ u \in T_C \ , \ v \in T_V \text{ and } \exists\ x \in V(u)\ s.t.\ (x, v) \in E(C) \\ (x, y) & \text{if } \ u \in T_C \ , \ v \in T_C \text{ and } \exists\ x \in V(u)\ \exists\ y \in V(v)\ s.t.\ (x, y) \in E(C) \end{cases}$$

Moreover, for a node u with $\lambda(u) = (x, y)$, we use $\lambda(u)_1$ to deonote x and $\lambda(u)_2$ to denote y. For the root node r, $\lambda(r)_1$ is chosen arbitrarily if r is a cycle node. If r is a vertex node, $\lambda(r)_1 = r$. Note that $\lambda(r)_2$ is undefined.

Image of the branching path P into tree T. Observe that all the vertices of a cycle in P must occur consecutively. Otherwise, P will have to retrace an edge, which is not allowed in a path. By replacing all the vertices of a cycle c in P by c, we get a path in T. This path will be denoted by $img(P)$.

Representative node and depth of a branching path. Let P be a path in the simple cactus C such that $img(P) = n_1 n_2 \ldots n_m$. The representative node of P, denoted by $rep(P)$, is the node with minimum depth in $img(P)$. The depth of P, denoted by $depth(P)$, is the depth of $rep(P)$.

Classification and decomposition of a branching path. Suppose we have a branching path $P = p_1 p_2 \ldots p_t$ with $img(P) = n_1 n_2 \ldots n_m$. such that $rep(P) = n_r$. We classify the branching path P as follows:

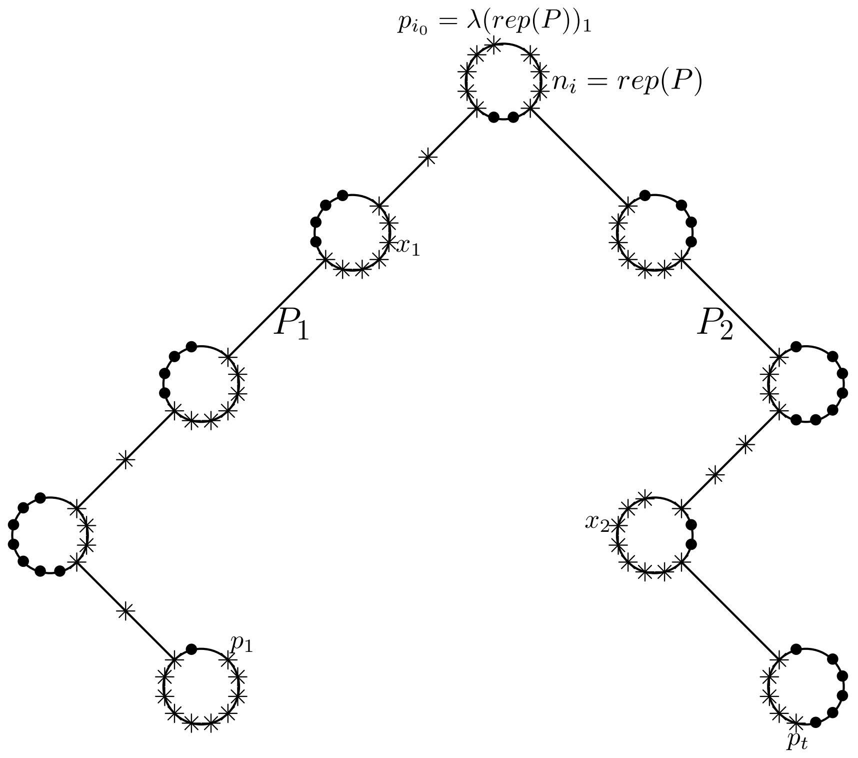

Figure 6: When $\lambda(rep(P)) \in V(P)$, P can be decomposed as $p_1P_1p_{i_0}P_2p_t$ where $p_{i_0} = \lambda(rep(P))_1$. P is an upper branching path . Asterisk marked vertices are part of P and textbullet marked vertices do not participate in P.

- $n_r \in T_C$. Let $p_{i_1}p_{i_1+1}\dots p_{i_2}$ be the part of P present in node n_i, *i.e.*, $V(P) \cap V(n_i) = p_{i_1}p_{i_1+1}\dots p_{i_2}$.
  - If $\exists i_0 : i_1 \le i_0 \le i_2$ *s.t.* $p_{i_0} = \lambda(n_i)_1$, P is called a upper branching path. P can be decomposed as $p_1P_1p_{i_0}P_2p_t$.
  - Otherwise, P is called a lower branching path. P can be decomposed as $p_1P_1p_{i_1}P_0p_{i_2}P_2p_t$.
- If $n_r \in T_V$, then also P is called an upper branching path. P can be decomposed as $p_1P_1n_iP_2p_t$.

The above three cases have been described pictorially in Figure 6, 7 and 8 respectively.

Partial order on the set of branching paths. We now define a partial ordering, $\lhd$, on the set of branching paths BP. Let P_a and P_b be two branching paths.

- If P_a is a subpath of P_b, then $P_b \lhd P_a$. **[Case 1]**
- If $rep(P_a)$ lies in a subtree rooted at $rep(P_b)$ and $rep(P_a) \neq rep(P_b)$, then $P_b \lhd P_a$. **[Case 2]**
- If $rep(P_a) = rep(P_b)$, then:

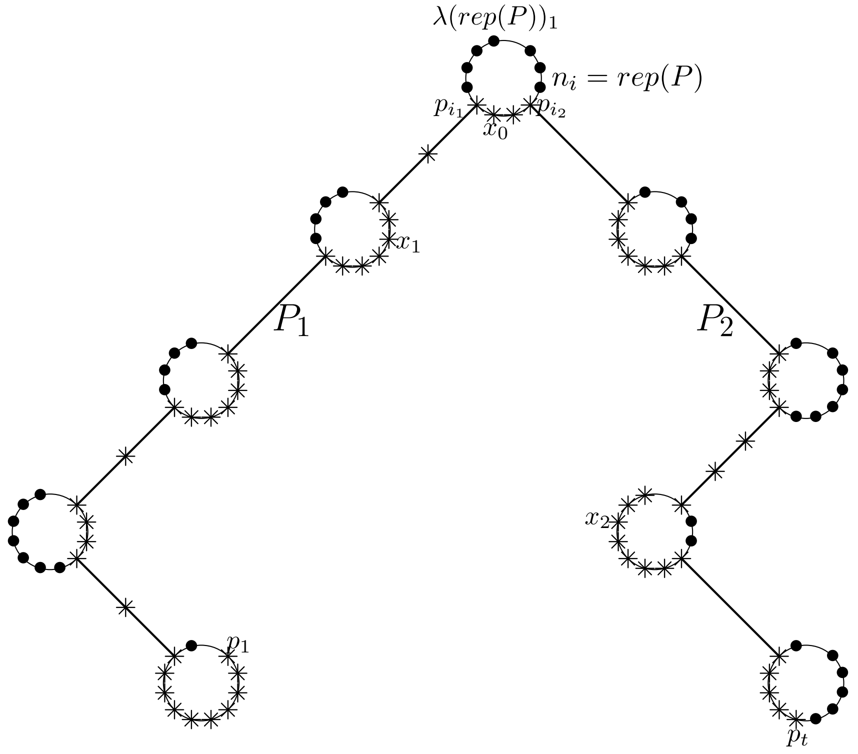

Figure 7: When $\lambda(rep(P)) \notin V(P)$, P can be decomposed as $p_1 P_1 p_{i_1} P_0 p_{i_2} P_2 p_t$ where p_{i_1} and p_{i_2} are the extreme vertices of P occurring in $rep(P)$. P is a lower branching path . Asterisk marked vertices are part of P and textbullet marked vertices do not participate in P.

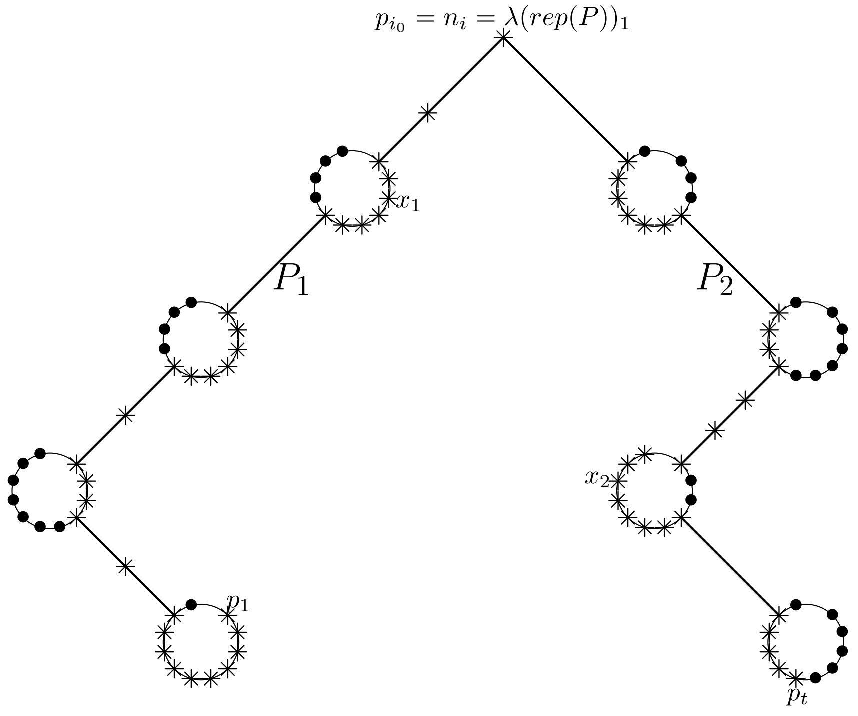

Figure 8: When $rep(P) \in T_V$, P can be composed as $p_1 P_1 n_i P_2 p_t$ where $p_{i_0} = \lambda(rep(P))$. This case is similar to the previous case. P is an upper branching path .

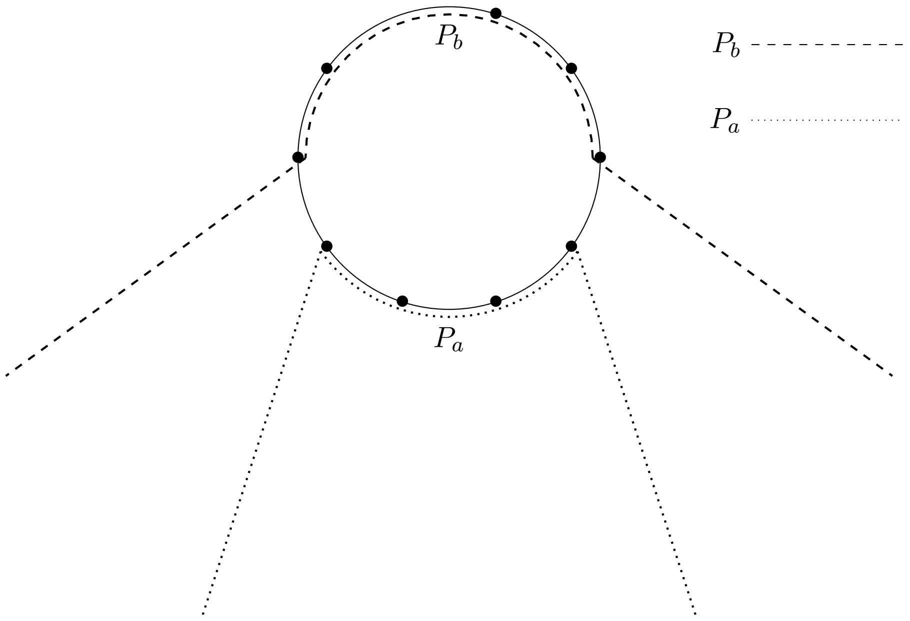

Figure 9: When both P_a and P_b have same representative node but P_b is a upper branching path whereas P_a is a lower branching path , $P_b \lhd P_a$.

- If P_a is a lower branching path and P_b is a upper branching path, then $P_b \lhd P_a$ (Figure 9). **[Case 3a]**
- If both P_a and P_b are lower branching path and the set of vertices of P_a present in $rep(P_a)$ is a strict subset of the set of vertices of P_b present in $rep(P_b)(= rep(P_a))$, *i.e.*, $V(P_a) \cap V(rep(P_a)) \subsetneq V(P_b) \cap V(rep(P_b))$, then $P_b \lhd P_a$ (Figure 10). **[Case 3b]**

Remark 34 *$\lhd$ is a partial ordering.*

Definition 35 *A branching path P is called a maximal branching path if there does not exist a branching path P' such that $P \lhd P'$.*

Starting with a branching path P_0, our aim is to move towards a maximal branching path P.

Lemma 36 *Given a branching path P_a, we can find, in polynomial time, a branching path P_b such that $P_a \lhd P_b$ if such a branching path exists.*

Proof:
A branching path P_b such that $P_a \lhd P_b$ must satisfy at least one of the four rules.

Case 1 Suppose there exists a branching path P_b which is a subset of $P_a = p_1p_2 \dots p_t$. To find P_b, for each $i \leq j$, check if $p_ip_{i+1} \dots p_j$ is a branching path.

Case 2 Suppose there exists a branching path P_b such that $rep(P_b)$ lies in the subtree rooted at $rep(P_a)$ and $depth(P_a) < depth(P_b)$, *i.e.*, $rep(P_b)$ is a strict descendant of $rep(P_a)$. For any child node of $rep(P_a)$ in T, say n_r, let C_r be the sub-cactus corresponding to the subtree rooted at n_r. Find if C_r contains a branching path using Lemma 31. Repeat this process for each child node of $rep(P_a)$.

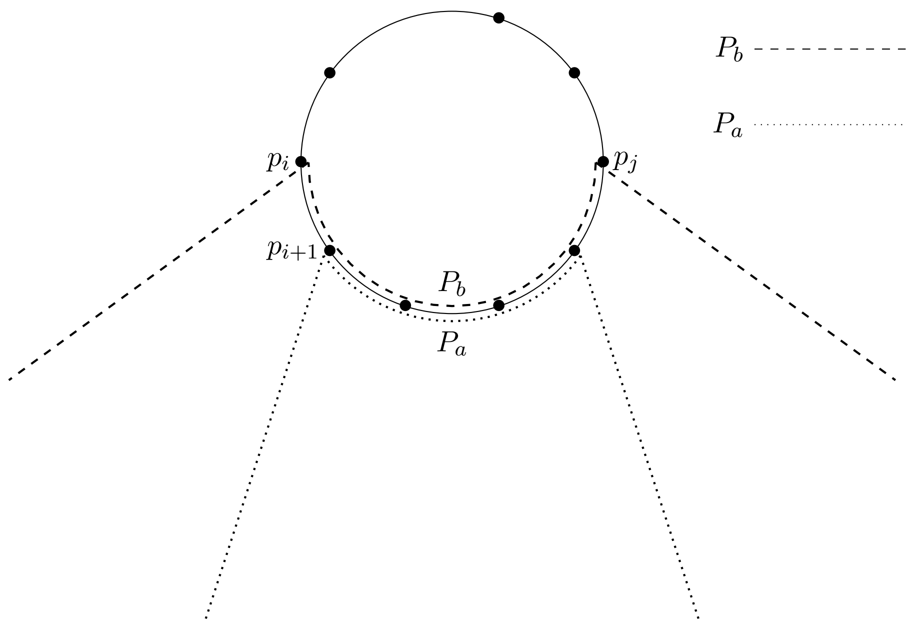

Figure 10: When both P_a and P_b are lower branching path with the same representative node, n, such that vertices of P_a in n is a strict subset of vertices of P_b in n, then $P_b \lhd P_a$.

Case 3a Suppose there is a branching path P_b of same depth as that of P_a such that P_b is a lower branching path and P_a is an upper branching path. We assume that there does not exist a branching path P' such that $P_a \lhd P'$ by Case 1 or Case 2. Else, we can find a new branching path P' such that $P_a \lhd P'$ (by Case 1 or Case 2).

Let T' be the tree rooted at $rep(P_a)$. Obviously, P_b must be a branching path in T' which does not pass through $\lambda(rep(P_a))_1$. See Figure 9. To find P_b, use Lemma 31 to find a branching path in the graph $T' - \{\lambda(rep(P_a))_1\}$.

Case 3b Suppose P_a is a lower branching path and there exists a lower branching path P_b with the same representative node, n, such that the set of vertices of P_b in n is a strict subset of the set of vertices of P_a in n, *i.e.*, $V(P_b) \cap V(n) \subsetneq V(P_a) \cap V(n)$. Like the previous case, we assume that there does not exist a branching path P' such that $P_a \lhd P'$ by Case 1, Case 2 or Case 3a. Else, we can find a new branching path P' such that $P_a \lhd P'$ (by Case 1, Case 2 or Case 3a).

Suppose $P_a = p_1 p_2 \dots p_t$ such that the part of P in its representative node is $p_i p_{i+1} \dots p_j$, *i.e.*, $V(P_a) \cap V(n) = \{p_i, p_{i+1}, \dots, p_j\}$. Let T' be the subtree rooted at n. As shown in Figure 10, P_b must lie in the subgraph $T'' := (T' - V(n)) \cup \{p_i, p_{i+1}, \dots, p_j\}$. However, P_b can not pass through both p_i and p_j. To find P_b, use Lemma 31 to a branching path either in the graph $T'' - \{p_i\}$ or in the graph $T'' - \{p_j\}$.

Each of the above four steps can be done in polynomial time. □

If C is a connected component of $H - N$ such that $C \cup N$ has an even cycle or $C \cup N$ has fewer connected components than in N, then, by Lemmas 31 and 36, we can compute a maximal branching path in polynomial time.

Properties of a maximal branching path. Recall that a port vertex is a vertex in C which has neighbors in N. Denote the set of port vertices by B. For a path P in C, let E_P denote the endpoints of P and B_P denote the set of port vertices in P, *i.e.*, $B_P := B \cap V(P)$.

Lemma 37 *If P is a maximal branching path, $E_P \subseteq B_P$.*

Proof: Follows by the definition of branching paths. In fact, Lemma 37 is true for all branching paths. □

Lemma 38 *If P is a maximal branching path, $|B_P| \leq 3$.*

Proof: Suppose P is such that $|B_P| > 3$. Let $b_1, b_2, b_3 \in B_P$ be three vertices which are consecutive when ordered by P. By the construction proposed in Lemma 27 and using Corollary 28, it's clear that the new branching path P' computed is a strict subpath of P. Hence, by Case 1, $P \lhd P'$ contradicting the fact that P is maximal. □

If $P = p_1 p_2 \ldots p_t$ is a lower branching path, P can be decomposed as $p_1 P_1 p_{i_1} P_0 p_{i_2} P_2 p_t$. We define three sets X_0, X_1 and X_2 for P_0, P_1 and P_2 respectively:

$$\begin{aligned}
X_0 =& \{p \mid p \in V(P_0),\ \text{there exists a path from } p \text{ to } N \text{ in the graph } C \setminus (P_0 \cup \{p_{i_1}, p_{i_2}\}) \cup \{p\} \cup N\ \}\\
X_1 =& \{p \mid p \in V(P_1),\ \text{there exists a path from } p \text{ to } N \text{ in the graph } C \setminus (P_1 \cup \{p_1, p_{i_1}\}) \cup \{p\} \cup N\ \}\\
X_2 =& \{p \mid p \in V(P_2),\ \text{there exists a path from } p \text{ to } N \text{ in the graph } C \setminus (P_2 \cup \{p_{i_2}, p_t\}) \cup \{p\} \cup N\ \}
\end{aligned}$$

Similary, for a upper branching path $P = p_1 p_2 \ldots p_t$ which can be decomposed as $p_1 P_1 p_{i_0} P_2 p_t$, we define X_1 and X_2 analogously.

$$\begin{aligned}
X_1 =& \{p \mid p \in V(P_1),\ \text{there exists a path from } p \text{ to } N \text{ in the graph } C \setminus (P_1 \cup \{p_1, p_{i_0}\}) \cup \{p\} \cup N\ \}\\
X_2 =& \{p \mid p \in V(P_2),\ \text{there exists a path from } p \text{ to } N \text{ in the graph } C \setminus (P_2 \cup \{p_{i_0}, p_t\}) \cup \{p\} \cup N\ \}
\end{aligned}$$

The best way to think as X_1, X_2 and X_0 as are "points of escape" from P_1, P_2 and P_0 to N respectively.

Corollary 39 *Let P be a lower branching path whose decomposition is $p_1 P_1 p_{i_1} P_0 p_{i_2} P_2 p_t$.*

- *For all vertices $v \in X_0$, v is connected to a port vertex $b_v \in B \setminus \{p_{i_1}, p_{i_2}\}$ by a path which does not intersect $P_0 \cup \{p_{i_1}, p_{i_2}\}$.*
- *For all vertices $v \in X_1$, v is connected to a port vertex $b_v \in B \setminus \{p_1, p_{i_1}\}$ by a path which does not intersect $P_1 \cup \{p_1, p_{i_1}\}$.*
- *For all vertices $v \in X_2$, v is connected to a port vertex $b_v \in B \setminus \{p_{i_2}, p_t\}$ by a path which does not intersect $P_2 \cup \{p_{i_2}, p_t\}$.*

Let P be a upper branching path whose decomposition is $p_1 P_1 p_{i_0} P_2 p_t$.

- *For all vertices $v \in X_1$, v is connected to a port vertex $b_v \in B \setminus \{p_1, p_{i_0}\}$ by a path which does not intersect $P_1 \cup \{p_1, p_{i_0}\}$.*

- *For all vertices $v \in X_2$, v is connected to a port vertex $b_v \in B \setminus \{p_{i_0}, p_t\}$ by a path which does not intersect $P_2 \cup \{p_{i_0}, p_t\}$.*

Lemma 40 *Let P be a maximal branching path. Then $|X_1| \leq 1$, $|X_2| \leq 1$ and $|X_0| \leq 1$.*

Proof: Suppose X_1 contains at two vertices. Let $p_a, p_b \in X_1$ be two vertices which are consecutive when ordered by P_1. By Corollary 39, p_a is connected to a port vertex b_a by path P_a and p_b is connected to a port vertex b_b by path P_b. Consider the graph $P_a \cup P_b \cup P_1 \cup \{b_1\}$ connecting $\{b_a, b_b, p_1\}$ which satisfies the conditions of Lemma 27. Hence, there exists a branching path P' in $P_a \cup P_b \cup P_1 \cup \{p_1\}$. If P is a lower branching path, depth of P' is strictly greater than that of P and by Case 2, $P \lhd P'$ contradicting the maximality of P. If P is a upper branching path, either P' passes through $rep(P)$ or it does not. If P' pass through $rep(P)$, it does not pass through $\lambda(rep(P))(= p_{i_0})$. Hence, P' is a lower branching path and by Case 3a, $P \lhd P'$ contradicting the maximality of P. If P' does not pass through $rep(P)$, depth of P' must be strictly greater than depth of P and by Case 2, $P \lhd P'$ contradicting the maximality of P. In any case, we have found a new branching path P' with depth greater than that of P which contradicts the maximality of P.

We can argue similarly for X_2 as well.

For X_0, let $p_a, p_b \in X_0$ be two vertices which are consecutive when ordered by P_0 ($i_1 \leq a \leq b \leq i_2$, *i.e.*, p_a is closer to p_{i_1} than p_b). By Corollary 39, p_a is connected to a port vertex b_a by path P_a and p_b is connected to a port vertex b_b by path P_b. Consider the graph $P_a \cup P_b \cup P_1 \cup P_0 \cup \{p_1, p_{i_0}\}$ connecting $\{b_a, b_b, p_1\}$ which satisfies the conditions of Lemma 27. Hence, there exists a branching path P' in $P_a \cup P_b \cup P_1 \cup P_0 \cup \{p_1, p_{i_0}\}$. Since $V(P') \cap V(rep(P'))$ is either $p_{i_1} p_{i_1+1} \dots p_a$ or $p_{i_1} p_{i_1+1} \dots p_b$ or $p_a p_{x_a+1} \dots p_b$ each of which is a strict subset of $p_{i_1} p_{i_1+1} \dots p_{i_2} = V(P) \cap V(rep(P))$, by Case 3b, $P \lhd P'$ contradicting the fact that P is maximal. $\square$

Lemma 41 *Let P be a maximal branching path. There exists an optimal solution S such that $S \cap V(P_1) \subseteq X_1$, $S \cap V(P_2) \subseteq X_2$ and $S \cap V(P_0) \subseteq X_0$.*

Proof: Let $X_1 = \{x_1\}$ and $v \in V(P_1) \setminus X_1$ be a vertex in S. Suppose v lies between p_1 and x_1. If all branching paths passing through v pass through p_1 or x_1 as well, replace v by two vertices p_1 and x_1 in S. Else, there exists a branching path P' entering P at y and leaving P at z such that y is between p_1 and v; and z is between v and x_1. By Case 2, $P \lhd P'$. When v lies between x_1 and p_{i_1} (p_{i_0}) for a lower (upper, respectively) branching path P, we can argue similarly using Case 2 and Case 3a. This contradicts the fact that P is maximal.

We can argue similarly for X_2, P_2 as well.

Let $X_0 = \{x_0\}$ and $v \in V(P_0) \setminus X_0$ be a vertex in S. Suppose v lies between p_{i_1} and x_0. If all branching paths passing through v pass through p_{i_1} or x_0 as well, replace v by two vertices p_{i_1} and x_1 in S. Else, there exists a branching path P' entering P at y and leaving P at z such that y is between p_{i_1} and v; and z is between v and x_0. By Case 3b, $P \lhd P'$. When v lies between x_0 and p_{i_2} for a lower branching path P, we can argue similarly. This contradicts the fact that P is maximal. $\square$

Lemma 42 *Let P be a maximal branching path. There exists an optimal solution S such that*

$$S \cap V(P) \subseteq \begin{cases} X_0 \cup \{p_{i_1}, p_{i_2}\} & \textit{if } P \textit{ is a lower branching path} \\ \{p_{i_0}\} & \textit{if } P \textit{ is an upper branching path} \end{cases}$$

Proof: Let $X_1 = \{x_1\}, X_2 = \{x_2\}$, $X_0 = \{x_0\}$ (By Lemma 40). Using Lemma 41, we can say that there exists a solution that S such that $S \cap V(P) \subseteq \{p_1, x_1, p_{i_1}, x_0, p_{i_2}, x_2, p_t\}$ when P is a lower branching path and $S \cap V(P) \subseteq \{p_1, x_1, p_{i_0}, x_2, p_t\}$ when P is a upper branching path. Suppose $x_1 \in S$.

If P is a lower branching path, any branching path P' passing through x_1 must also pass through p_{i_1} or $P \lhd P'$ by Case 2 contradicting the fact that P is a maximal branching path.

Suppose $P = p_1 p_2 \dots p_t$ is a upper branching path. Let p_{i_1} and p_{i_2} be the points of incidence of P in $rep(P)$ $(i_1 \leq i_0 \leq i_2)$, *i.e.*, $V(P) \cap V(rep(P)) = p_{i_1} \dots p_{i_0} \dots p_{i_2}$. Any branching path P' passing through x_1 must also pass through p_{i_1} since P is maximal (Case 2). If P' is a upper branching path, x_1 can be replaced by p_{i_0}. P' can't be a lower branching path because if P' is a lower branching path, $P \lhd P'$ by Case 3a contradicting the fact the P is maximal.

The same argument applies to p_1 as well.

Similarly, we can argue for x_2 and p_t as well. □

Lemma 43 *Let P be a maximal lower branching path. There exists an optimal solution S such that $S \cap V(P) \subseteq \{p_{i_1}, p_{i_2}\}$.*

Proof: Let $X_0 = \{x_0\}$ (By Lemma 40). By Lemma 42, we can say that there exists a solution S such that $S \cap V(P) \subseteq \{x_0, p_{i_1}, p_{i_2}\}$. Suppose $x_0 \in S$. Let $P = p_1 \dots p_{i_1} \dots p_i \dots p_{i_2} \dots p_t$ be such that $p_i = x_0$. If a branching path passing through p_i also passes through p_{i_1} or p_{i_2}, we can replace p_i by p_{i_1} and p_{i_2} in S. Else, there exists a branching path P' is such that it enters P at $p_{i'_1}$ and leaves P at $p_{i'_2}$ such that $i_1 < i'_1 \leq i \leq i'_2 \leq i_2$ or $i_1 \leq i'_1 \leq i \leq i'_2 < i_2$. Then, by Case 3b, $P \lhd P'$ which contradicts the fact that P is maximal. □

Starting from the branching path P_0 obtained in Lemma 31, we can repeatedly apply Lemma 36 till we obtain a maximal branching path P' which satisfies Lemma 37, 38, 40, 41, 42 and 43. Then there exists an optimal solution S such that $S \cap V(P) \subseteq \{p_{i_1}, p_{i_2}\}$ if P is a lower branching path and $S \cap V(P) \subseteq \{p_{i_0}\}$ if P is an upper branching path.

If P is a maximal lower branching path, we branch in the following way: a) Deleting p_{i_1} from C and adding it to the solution set; b) Deleting p_{i_2} from C and adding it to the solution set; or c) Moving P to S. If P is a maximal upper branching path, we branch in the following way: a) Deleting p_{i_0} from C and adding it to the solution set; or b) Moving P to S.

Note that P can be such that $P \cup S$ does not have any even cycles but P has adjacencies to more than one connected component of S. In this case, moving a vertex to the final solution set might not be the optimal step. Instead, we move the whole of P to S as this step has a zero cost associated with it. The jusfication for this step is provided when we discuss the proof of correctness of this algorithm.

As long as a connected component C satisfies the conditions of Lemma 31, we branch on the vertices of a "maximal" branching path P of C. We do this for all connected components of $H - N$. At the end, all the connected components C are such that: a) $C \cup N$ does not have even cycles; and b) C has adjacencies to only one connected component of N.

Further more, using Lemma 27, the following corollary follows:

Corollary 44 *If a connected component C is such that $C \cup N$ does not have any even cycles and C has adjacencies to only one connected component of N, C has at most two edges going to N,* i.e., $|B| \leq 2$.

4.2 Second Phase

At the beginning of the second phase, we have the following facts: for every connected component C of $H - N$, $C \cup N$ does not have any even cycles and all edges from C to N are incident on the same connected component of N.

The even cycles still left to be dealt with are those which pass through multiple connected components of $H - N$. Also, if a connected component C of $H - N$ is such that there exists only one edge from C to N, C can be ignored because it can never participate in any such cycle.

The following corollary follows directly from Corollary 44:

Corollary 45 *From every connected component C of $H - N$, there are at most two edges going from C to N.*

Recall that B_C denotes the set of port vertices of a connected component C of $H - N$.

Lemma 46 *There exists an optimal solution S_0 such that for every connected component C of $H - N$, $S_0 \cap V(C) = \varnothing$ or $S_0 \cap V(C) = \{x\}$ for any $x \in B_C$.*

Proof: Let S_0' be an optimal solution. Let C be some connected component of $H - N$ and $x \in B_C$. Suppose $v \in V(C) \cap V(S_0)$. Note that $|B_C| \leq 2$. If $v \notin B_C$, any even cycle passing through v also passes through x. Hence, v can be replaced by x in S_0'. □

Lemma 47 *Let $Q_1, Q_2, \ldots, Q_l$ be pairwise vertex disjoint paths in a connected graph G'. Then, for some $i, j \in [l]$, there exists a path Q joining Q_i and Q_j which is vertex disjoint from Q_k for all $k \in [l] \setminus \{i, j\}$.*

Proof: For some $i \in [l]$, let u be a vertex of Q_i. Let v be the nearest vertex from u which belongs to Q_j for some $j \in [l] \setminus \{i\}$. Let the shortest path connecting u and v be $Q = uv_1v_2 \ldots v_tv$. Let v_a be the last vertex of P in P_i, *i.e.*, $v_a \in v(Q_i)$ and for all $b > a : v_b \notin V(Q_i)$. Then the path $v_av_{a+1} \ldots v_tv$ joining Q_i and Q_j is the required path. □

Lemma 48 *If there's a cycle in H passing through l ($l \geq 3$) connected components of $H - N$, then there is a cycle passing through at most $l - 1$ connected components of $H - N$.*

Proof: Suppose a cycle c passing through l connected components $C_1, C_2, \ldots, C_l$ of $H - N$. For $i \in [l]$, let c_i be the part of c in C_i. c can then be decomposed as $c_1n_1c_2n_2 \ldots c_ln_l$ where $n_1, n_2, \ldots, n_l$ are pairwise disjoint from each other. See Figure 11. Here, $n_1, n_2, \ldots, n_l$ lie in the same connected component of N because edges from C_i's are incident in the same connected component of N. Also, all c_i's lie in different connected components of $H - N$ because each connected component C of $H - N$ has at most two edges from C to N.

By Lemma 47, for some n_i and n_j ($i < j$), there exists a path s joining n_i and n_j which is disjoint from n_k for all $k \in [l] \setminus \{i, j\}$. Let $n_i = u_i \ldots u_a \ldots v_i$ and $n_j = u_j \ldots u_b \ldots v_j$ be such that s is of the form $u_a \ldots u_b$. Then, the cycle $c' = c_1n_1 \ldots c_i\ u_i \ldots u_a\ s\ u_b \ldots v_j\ c_{j+1} \ldots n_l$ bypasses $c_{i+1}, c_{i+2}, \ldots, c_j$ and hence passes through at most $l - 1$ connected components of $H - N$. □

Corollary 49 *If there's a cycle in H passing through l ($l \geq 3$) connected components of $H - N$, there is a cycle passing through at most two connect components of $H - N$.*

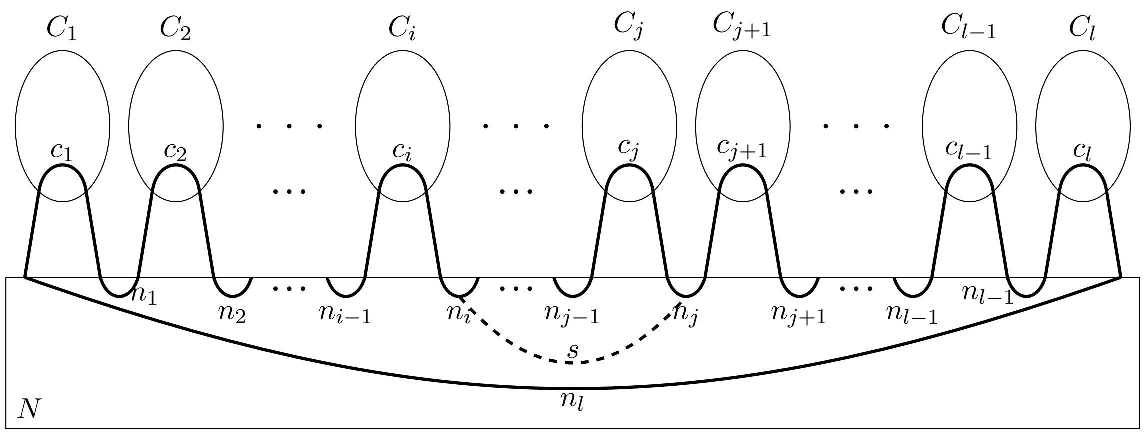

Figure 11: Thick line indicates the original cycle c. The intermediate path is substituted by the dashed path s.

Lemma 50 *If there's a cycle passing through two connected components C_1 and C_2 of $H - N$, there is an even cycle in $C_1 \cup C_2 \cup N$.*

Proof: Suppose a cycle c passes through only C_1 and C_2. If c is even, then the claim is trivially true. So assume that c is an odd cycle. Similar to the previous lemma, for $i = 1, 2$, let c_i be the part of c occurring in C_1. c can then be decomposed as $c_1 n_1 c_2 n_2$. By Lemma 47, there exists a path s joining n_1 and n_2 in N. There are now two cycles: one in $c_1 \cup n_1 \cup n_2 \cup s$ and the other in $c_2 \cup n_1 \cup n_2 \cup s$ both of which have s in common. By Lemma 6, there is an even cycles in $c_1 \cup c_2 \cup N$. □

Let $C_1, C_2 \ldots C_l$ be the different connected components of $H - N$. Define $C_{ij} = C_i \cup C_j$ and B_i to be the set of port vertices of C_i. Given some i_0 and j_0, any even cycle passing in $C_{i_0 j_0} \cup N$ must pass through all the vertices of $B_{i_0} \cup B_{j_0}$. As $|B_{i_0}| \leq 2$ and $|B_{j_0}| \leq 2$, by Lemma 46, we have a two way branching algorithm to delete even cycles in $C_{i_0 j_0}$. We do this for all possible pairs (C_i, C_j).

If there are no even cycles in $C_i \cup C_j \cup N$ for all $i, j \in [l]$, by Lemma 50, there are no cycles passing through two connected components of $H - N$. Hence, by Lemma 48, no cycle passes through more than two connected components of $H - N$. Even cycles passing through two connected components have already been dealth with. Also, even cycles passing through one connected component have also been dealth with. Hence, there's no even cycle in the graph.

4.3 Analysis

Time complexity of the algorithm. To analyze the time complexity of the algorithm, we look at the measure

$$\mu = k + \text{number of connected components of } N$$

At the beginning of the algorithm, $\mu \leq 2k$. First phase has a branching factor of three whereas second phase has a branching factor of two. Overall, it's a three way branching algorithm. Hence, time complexity of DISJOINT-ECT is $\mathcal{O}^*(3^{2k}) = \mathcal{O}^*(9^k)$.

By Theorem 16, IC-ECT will have time complexity $\mathcal{O}^*(10^k)$. Hence, ECT can be solved in time $\mathcal{O}^*(10^k)$

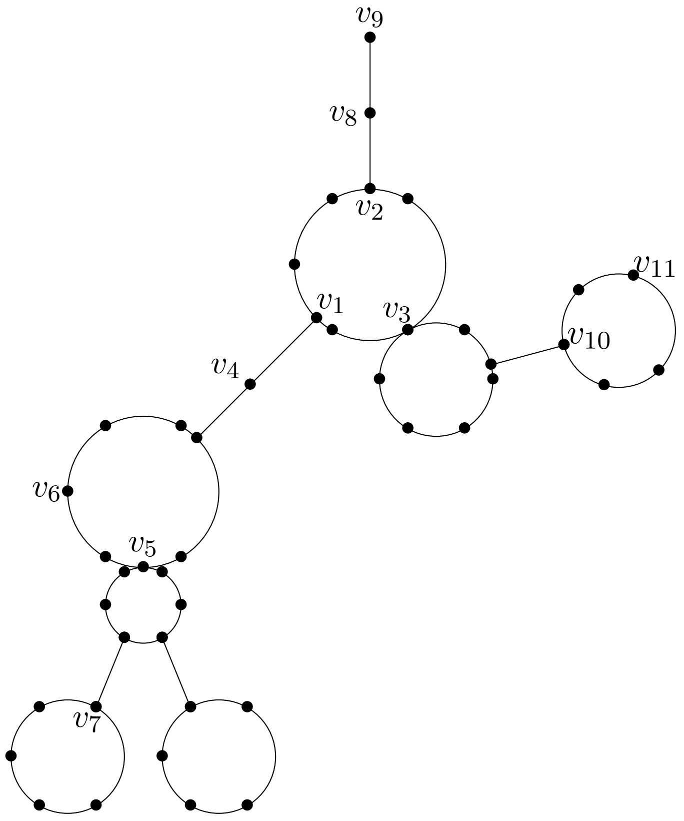

Figure 12: An odd cactus. $v_1, v_2, v_4, v_6, v_7, v_8, v_9, v_{10}, v_{11}$ are unique vertices and v_3, v_5 are non-unique vertices. $v_1, v_2, v_3, v_5, v_6, v_7, v_{10}, v_{11}$ are cycle vertices and v_4, v_8, v_9 are non-cycle vertices.

4.4 Transformation from a cactus to a simple cactus.

Definition 51 *A vertex of a cactus is called a unique vertex if it belongs to at most one cycle. A non-unique vertex of a cactus is a vertex which is not a unique vertex.*

Non-unique vertices are the vertices which are common to two or more cycles in a cactus. See Figure 12 for an illustration.

Remark 52 *A simple cactus is a cactus each of whose vertex is a unique vertex.*

If some connected component C of $H - N$ is such that it is not a simple cactus, we describe a transformation that converts C to a simple cactus C' such that the algorithm described above still works.

We first give an informal idea of construction by means of Figure 13. As shown in the figure, let v be a non-unique vertex occurring in multiple cycles of C. Let the blocks of C in which v occurs be $b_1, b_2, \ldots, b_5$. Then, for each block b_i, we keep a copy of v - say v_i. Furthermore, we connect each of these v_i's to a new vertex v'. The idea is to preserve the parity of paths going *via.* v into different blocks: a path going from b_i to b_j *via.* v would now traverse following sequence of vertex: $v_i v' v_j$. If a path *via.* v remains in a single block b_i, v is simply replaced by v_i in the new path.

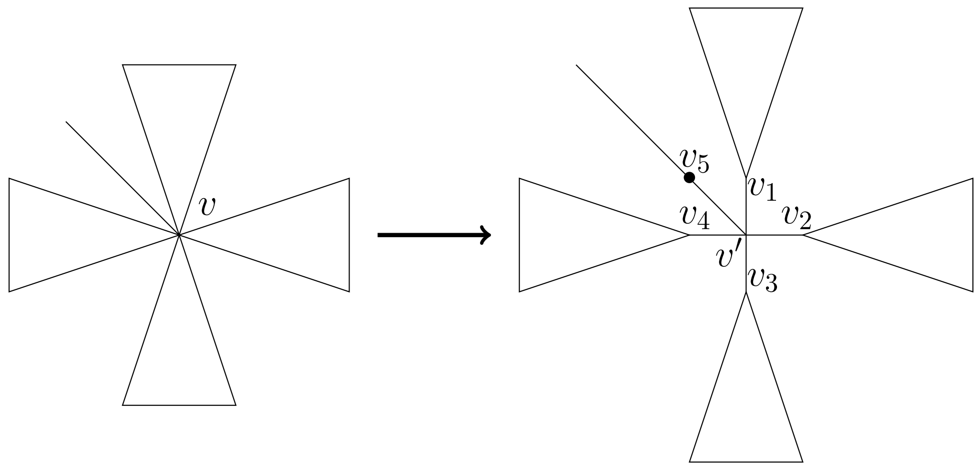

Figure 13: Transformation of C to C'

Also, if v is a port vertex, only one of $v_1, v_2, \ldots, v_5$ will be a port vertex. v' can not be a port vertex since it would not preserve the parity of branching paths. If all of v_i's are port vertices, one single branching path would have multiple images. Hence, one of v_i's is chosen arbitrarily to be a port vertex.

Suppose the block decomposition of C yields the set of blocks $B = \{b_1, b_2, \ldots, b_m\}$ [24]. We define two functions, $\mathit{blocks}\colon V(C) \to 2^B$ and $\mathit{block}\colon E(C) \to B$ which determines the set of blocks (block) a vertex (an edge, recpectively) occurs in.

$$\begin{aligned} \mathit{blocks}(v) &= \{b_i \mid v \in V(b_i)\} \\ \mathit{block}(e) &= b_i \text{ if } e \text{ occurs in block } b_i \end{aligned}$$

C' is defined formally as follows:

$$\begin{aligned} V(C') =& \big\{v_{i_1}, v_{i_2}, \ldots, v_{i_{m'}}, v' \mid v \in V(C), \mathit{blocks}(v) = \{b_{i_1}, b_{i_2}, \ldots, b_{i_m}\}\big\} \\ E(C') =& \big\{(u_i, v_i) \mid (u,v) \in E(C), \mathit{block}((u,v)) = b_i\big\} \ \cup \\ & \big\{(v_{i_1}, v'), (v_{i_2}, v'), \ldots, (v_{i_m}, v') \mid v \in V(C), \mathit{blocks}(v) = \{b_{i_1}, b_{i_2}, \ldots, b_{i_m}\}\big\} \end{aligned}$$

Remark 53 *If $v \in V(C)$ is a unique cycle vertex which occurs in only one block, $v' \in V(C')$ will have only one neighbor, which is v. Hence, v' won't be part of any branching path in C'.*

We can now describe the edges going across C' to N, *i.e.*, the port vertices of C'. If a port vertex v is such that $\mathit{blocks}(v) = \{b_{i_1}, b_{i_2}, \ldots, b_{i_m}\}$, only one of v_{i_j}'s can be a port vertex. Without loss of generality, assume it to be v_{i_j}. We say that v_{i_j} is the minimal image of v.

The edges, E, going across C' to N are:

$$E = \{(v_i, s) \mid v \in V(C), s \in V(N), (v,s) \in E(C \cup N), v_i \text{ is the minimal image of } v\}$$

Corollary 54 *Both C and C' have the same number of port vertices.*

Corollary 55 *If $v_i \in V(C')$ is a port vertex, $v \in V(C)$ is also a port vertex.*

Remark 56 *No port vertex of C' is of the form v'.*

Let $\mathcal{P}$ denote the set of paths of C and $\mathcal{P}'$ denote the set of paths of C' which start or end at vertices of the form v_i such that v_i is the minimal image of some $v \in V(C)$.

Lemma 57 *There exists a bijection between $\mathcal{P}$ and $\mathcal{P}'$ which is parity preserving.*

Proof: We present a construction mapping a path $P \in \mathcal{P}$ to $P' \in \mathcal{P}'$. Any unique vertex v of P are replaced by v_i where $blocks(v) = \{b_i\}$. For a non-unique vertex v of P such that P switches from block b_i to b_j, v is replaced by $v_i v' v_j$ in P'. If block remains the same while traversing v, say b_i, v is simply replaced by v_i.

It is easy to see that the above mapping is a parity preserving bijection. □

Let BP denote the set of branching paths in C and BP' denote the set of branching paths of C'. As an immediate application of Lemma 57, we have the following corollary:

Corollary 58 *There exists a parity preserving bijective mapping between BP and BP'.*

For a non-simple cactus C, we first change it to a simple cactus C' and determine the set of minimal vertex set (*i.e.*, either $\{p_{i_0}\}$ or $\{p_{i_1}, p_{i_2}\}$) by decomposition of a maximal branching path P' of C' For all vertices from this minimal set, we delete its inverse image from C and recursively call the new instance. The only remaining case is when we move the maximal branching path to N. In this case, we move the inverse image of P' in C to N.

References

[1] Bar-Yehuda, R., Geiger, D., Naor, J., and Roth, R. M. Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and bayesian inference. *SIAM journal on computing 27*, 4 (1998), 942–959.

[2] Becker, A., Bar-Yehuda, R., and Geiger, D. Random algorithms for the loop cutset problem. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence* (1999), Morgan Kaufmann Publishers Inc., pp. 49–56.

[3] Bender, M., and Farach-Colton, M. The lca problem revisited. In *LATIN 2000: Theoretical Informatics*, G. Gonnet and A. Viola, Eds., vol. 1776 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2000, pp. 88–94.

[4] Bodlaender, H. L. On disjoint cycles. *International Journal of Foundations of Computer Science 5*, 01 (1994), 59–68.

[5] Cao, Y., Chen, J., and Liu, Y. On feedback vertex set new measure and new structures. In *Algorithm Theory-SWAT 2010*. Springer, 2010, pp. 93–104.

[6] Chandrasekaran, K., Karp, R., Moreno-Centeno, E., and Vempala, S. Algorithms for implicit hitting set problems. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms* (2011), SIAM, pp. 614–629.

[7] Chen, J., Fomin, F. V., Liu, Y., Lu, S., and Villanger, Y. Improved algorithms for feedback vertex set problems. *Journal of Computer and System Sciences 74*, 7 (2008), 1188–1198.

[8] Chvatal, V. A greedy heuristic for the set-covering problem. *Mathematics of operations research 4*, 3 (1979), 233–235.

[9] Cygan, M., Nederlof, J., Pilipczuk, M., Pilipczuk, M., van Rooij, J. M., and Wojtaszczyk, J. O. Solving connectivity problems parameterized by treewidth in single exponential time. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on* (2011), IEEE, pp. 150–159.

[10] Dehne, F., Fellows, M. R., Langston, M. A., Rosamond, F. A., and Stevens, K. An o (2ˆ oˆ(ˆ kˆ) nˆ 3) fpt algorithm for the undirected feedback vertex set problem. In *COCOON* (2005), vol. 3595, Springer, pp. 859–869.

[11] Downey, R. G., and Fellows, M. R. Fixed-parameter tractability and completeness i: Basic results. *SIAM Journal on Computing 24*, 4 (1995), 873–921.

[12] Downey, R. G., and Fellows, M. R. *Parameterized complexity.* Springer Science & Business Media, 2012.

[13] Guo, J., Gramm, J., Hüffner, F., Niedermeier, R., and Wernicke, S. Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. *Journal of Computer and System Sciences 72*, 8 (2006), 1386–1396.

[14] Hartmanis, A. C. D. H. J., Henzinger, T., Leighton, J. H. N. J. T., and Nivat, M. Texts in theoretical computer science an eatcs series.

[15] Kakimura, N., Kawarabayashi, K.-i., and Kobayashi, Y. Erdös-pósa property and its algorithmic applications: parity constraints, subset feedback set, and subset packing. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms* (2012), SIAM, pp. 1726–1736.

[16] Kanj, I., Pelsmajer, M., and Schaefer, M. Parameterized algorithms for feedback vertex set. In *Parameterized and Exact Computation.* Springer, 2004, pp. 235–247.

[17] Karp, R. M. *Reducibility among combinatorial problems.* Springer, 1972.

[18] Kociumaka, T., and Pilipczuk, M. Faster deterministic feedback vertex set. *Information Processing Letters 114*, 10 (2014), 556–560.

[19] Lokshtanov, D., Narayanaswamy, N., Raman, V., Ramanujan, M., and Saurabh, S. Faster parameterized algorithms using linear programming. *ACM Transactions on Algorithms (TALG) 11*, 2 (2014), 15.

[20] Lokshtanov, D., Saurabh, S., and Sikdar, S. Simpler parameterized algorithm for oct. In *Combinatorial algorithms.* Springer, 2009, pp. 380–384.

[21] Misra, P., Raman, V., Ramanujan, M., and Saurabh, S. Parameterized algorithms for even cycle transversal. In *Graph-Theoretic Concepts in Computer Science* (2012), Springer, pp. 172–183.

[22] Raman, V., Saurabh, S., and Subramanian, C. Faster fixed parameter tractable algorithms for finding feedback vertex sets. *ACM Transactions on Algorithms (TALG) 2*, 3 (2006), 403–415.

[23] Reed, B., Smith, K., and Vetta, A. Finding odd cycle transversals. *Operations Research Letters 32*, 4 (2004), 299–301.

[24] Stein, C., Cormen, T., Rivest, R., and Leiserson, C. *Introduction to algorithms.* MIT press, 2009.

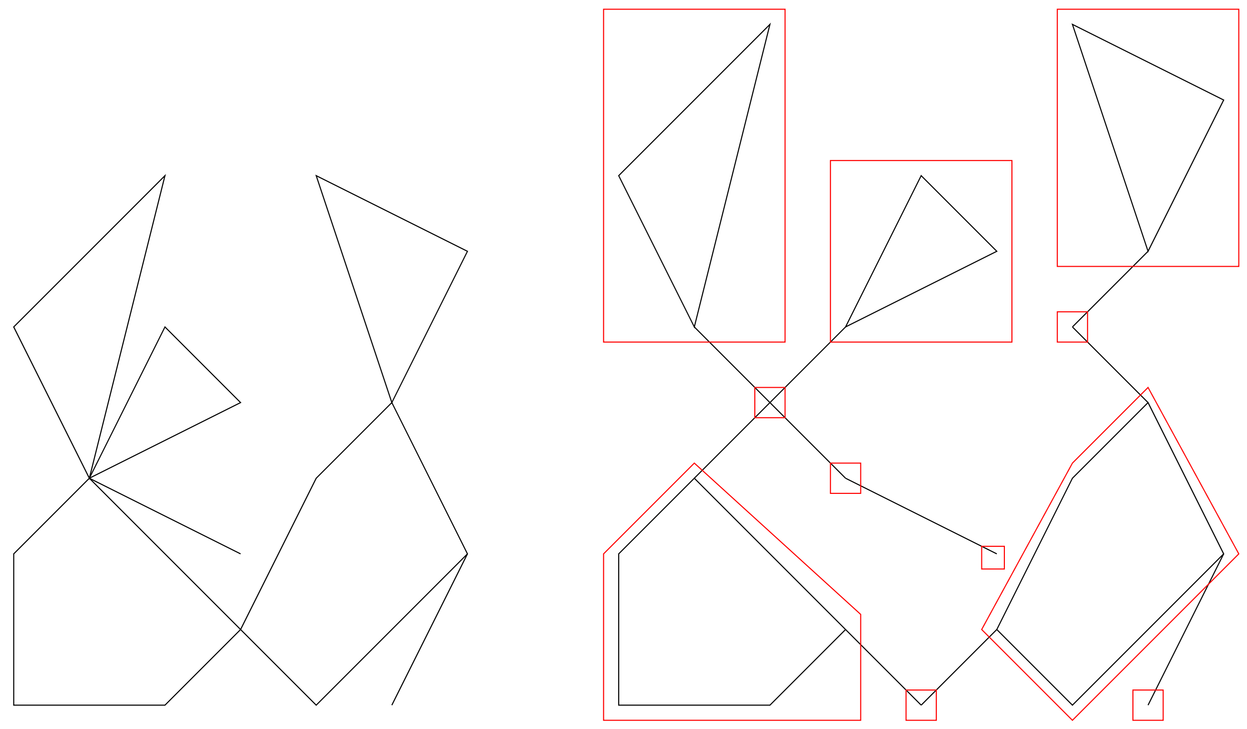

Figure 14: Graph transformation of $C \to C' \to T$. The first graph is C. The second graph (omitting red lines) is C'. The red boxes indicate each of the node of the tree T. The black edges running across the red boxes are the edges of tree T.