MITIT 2026 Spring Invitational Editorials: Finals

The MITIT Organizers

April 5, 2026

A Edit Distance Parity

Problem Idea: Zixiang Zhou
Problem Preparation: Adhitya Mangudy Venkata Ganesh
Analysis By: Adhitya Mangudy Venkata Ganesh

Suppose we have two sequences $a[1], \ldots, a[n]$ and $b[1], b[2], \ldots, b[m]$, and the edit distance between $a[1], \ldots, a[i]$ and $b[1], \ldots, b[j]$ is $e[i][j]$.
Edit distance satisfies the following property

$$e[i][j] = \begin{cases} e[i-1][j-1] \text{ if } a[i] = b[j] \\ \min(e[i-1][j], e[i][j-1], e[i-1][j-1]) + 1 \text{ if } a[i] \neq b[j] \end{cases} \tag{A.1}$$

with base cases $e[0][j] = j, e[i][0] = i$.
The main observation is that all the values of $e[i][j]$ can be determined using their parities. The values for $e[i][0]$ and $e[0][j]$ are known, and as $e[i-1][j-1] \leq e[i][j-1]+1, e[i-1][j]+1$, the property in A.1 implies

$$e[i][j] = \min(e[i-1][j], e[i][j-1], e[i-1][j-1]) \text{ or } \min(e[i-1][j], e[i][j-1], e[i-1][j-1])+1$$

This can be used to compute the value of $e[i][j]$ by a DP using its parity to choose the correct answer.
After computing $e[i][j]$, using A.1 there will be some conditions of the form $a[i] = b[j]$ or $a[i] \neq b[j]$ depending on which one is true. If neither is true, there is no solution, and if both are true, then there is no condition on $a[i], b[j]$ from this. Now, the problem is reduced to a coloring problem with constraints of the form $a[i] = b[j]$ or $a[i] \neq b[j]$.
This can be solved by constructing a graph on $n + m$ vertices, which correspond to $a[1], a[2], \ldots, a[n], b[1], b[2], \ldots, b[m]$. Connect an edge between $a[i]$ and $b[j]$ if $a[i] = b[j]$, and compute the connected component formed by this. If any condition $a[i] \neq b[j]$ has $a[i], b[j]$ in the same connected component, there is no valid solution. Otherwise, assign a different number to each connected component, and this will give a valid pair of sequences.

B Fruit Blast

Problem Idea: Sam Zhang
Problem Preparation: Sam Zhang
Analysis By: Sam Zhang

Theorem B.1. *Let $x_1, \ldots, x_{N^2}$ be a level (so each $k \in [1, N]$ appears N times in x), and for each $1 \le k \le N^2$, let $c_k(y)$ be the number of indices $1 \le i \le k$ such that $x_i \equiv y \pmod N$. Then the level is winnable if and only if for all $1 \le k \le N^2$ and $1 \le y \le N$, we have $c_k(y) \le c_k(y-1) + 1$.*

Proof. First we show necessity. Consider any prefix $x_1, \ldots, x_k$ of the level. If the level is winnable, then in the decomposition into subsequences, each y in this prefix is either preceded by a copy of $y-1 \pmod N$ earlier in its subsequence, or is the first element of the subsequence (which can only occur for a single subsequence). Thus in the winnable case all but at most one copy of y is preceded by its own copy of $y-1$, so $c_k(y) \le c_k(y-1)+1$.

Next we show sufficiency. We use the following algorithm to construct a decomposition of x into the desired subsequences:

- Let S_k be the subsequence $k \bmod N, (k+1) \bmod N, \ldots, 1, \ldots, (k-1) \bmod N$. Consider each x_i in increasing order of i. For each i, if $c_i(x_i) = m$, we assign x_i to be the m-th element of the subsequence $S_{x_i - m + 1}$.

To prove the correctness of this algorithm, we need to show that for each subsequence, its elements are being assigned in the correct order (the first element assigned before the second element, and so on). Suppose that this algorithm properly assigns the elements $x_1, \ldots, x_{i-1}$, in such a way that a prefix of each subsequence S_k in the decomposition is filled in already; we want to show that when we assign x_i, its placement extends one of these subsequences.

Under the theorem condition, $c_i(x_i) \le c_i(x_i - 1) + 1$. Each copy of $(x_i - 1) \bmod N$ counted by $c_i(x_i - 1)$, each already placed in some subsequence (in the positions $1, \ldots, c_i(x_i - 1)$), "unlocks" the placement of a copy of x_i immediately afterwards in the same subsequence (except in the case of $c_i(x_i - 1) = N$, but that is not an issue because a strict inequality $c_i(x_i) < c_i(x_i - 1) + 1$ would hold in that case), in such a way that our algorithm will place the x_i's in the unlocked positions in the correct order. Moreover, this count of available positions for the value x_i does not count the subsequence S_{x_i} that starts with x_i, which is unlocked at the start. Thus, when the bound $c_i(x_i) \le c_i(x_i - 1) + 1$ holds, placing x_i following the algorithm will always give enough unlocked positions, so that by placing in an unlocked position, we ensure that the algorithm assigns subsequences in the correct order. □

Consider the following setup. There is a circle with $2N$ cells, and cells $1, 3, \ldots, 2N-1$ initially contain tokens, respectively labeled $1, \ldots, N$. In a single move, we move a token one cell clockwise, so that it is not located in the same cell as any other token. Using our theorem, given a level $x_1, \ldots, x_{N^2}$, we may interpret it as a sequence of moves (where the i-th move moves token x_i), so that the level is winnable if and only if this sequence of moves is valid and leads to the cells $N+1, N+3, \ldots, 3N-1 \pmod{2N}$ containing tokens. We can thus solve the problem by dynamic programming, where the state consists of the set of values $1 \le x \le 2N-1$ such that the cell x cells clockwise from token 1 contains a token. The runtime is $O\left(2^{2N} \cdot \text{poly}(N)\right)$, with the polynomial degree depending on implementation details.

C Tree Partition

Problem Idea: Zixiang Zhou
Problem Preparation: Zixiang Zhou
Analysis By: Zixiang Zhou

There are multiple solutions for this problem, and we only describe the author's solution. Throughout this analysis, let the *length* of a path refer to its number of vertices.

Subtask 1

This subtask rewards slow implementations of the subtask 2 solution, various other dynamic programming approaches, and possibly some well-implemented brute force solutions.

Subtask 2

First, observe that if the answer is nonzero, then there are at most $2\log_2(N+1)$ vertices of degree at least 3 in the tree, and at most $2\log_2(N+1)$ leaves.

The key observation for this subtask is that there are at most $O(N\log^2 N)$ paths in the tree with a power-of-two length. To see this, fix one endpoint of the path and its length. As we extend the path, we can only branch into multiple choices when we encounter a vertex of degree at least 3. Since there are only $O(\log N)$ such branching points, the number of paths with a fixed endpoint and length is $O(\log N)$, giving a total of $O(N\log^2 N)$.

One way to efficiently enumerate these paths in $O(N\log^2 N)$ time is to perform $O(\log N)$ DFS traversals from each leaf. During each DFS, we maintain the current root-to-vertex path, which allows us to efficiently identify ancestors at a power-of-two distance away.

Consider dynamic programming where $\mathrm{dp}[u]$ denotes the number of ways to partition the subtree of u into power-of-two length paths. Crucially, observe that the binary representation of the subtree size of u uniquely determines which path lengths must be used.

Group the power-of-two length paths by their LCA. To compute $\mathrm{dp}[u]$, iterate over all paths whose LCA is u. For each such path, check whether the $O(\log N)$ components formed after removing it from the subtree have compatible sizes (i.e., not sharing any bits), and combine the DP values accordingly.

The overall time complexity is $O(N\log^3 N)$. Some constant-factor optimizations may be necessary to pass the time limit, such as using a sparse table for $O(1)$ LCA queries.

Subtask 3

Recall the first observation from subtask 2 that if the answer is nonzero, then there are at most $4\log_2(N+1)$ vertices whose degree is not equal to 2. Call these vertices *special*. We

can reduce to $M \leq 4 \log_2(N+1)$ by contracting degree-2 vertices, so the compressed tree is essentially very small.

We would like to perform a similar dynamic programming as in subtask 2, but we cannot afford a state for every vertex in the original (uncompressed) tree. Observe that if we use a path whose endpoint is in the middle of a degree-2 chain, then it must be extended by another path, and this path is essentially forced until it reaches a special vertex or a leaf. If it also ends in the middle of a degree-2 chain, then the next path is forced, and so on. This motivates considering a *maximal sequence* of power-of-two length paths that extends in both directions until reaching special vertices (or leaves). By considering maximal sequences of paths between special vertices, we can reduce the DP to only $O(\#$ special vertices) states.

In more detail, we define two types of DP states over special vertices u:

- Let $\mathrm{dp}[u]$ be the number of ways to partition the subtree of u.
- Let $\mathrm{dp2}[u]$ be the number of ways to partition the subtree of u plus the chain up to u's nearest special ancestor p (not including p itself).

To compute $\mathrm{dp}[u]$, we enumerate all pairs of special vertices whose LCA is u, say (v, w), and compute the number of ways a maximal sequence could have endpoints near v and w. If the sequence ends right before covering v, we multiply by $\mathrm{dp}[v]$, and if it covers v, we multiply by dp2 of the children of v, and similarly for w. We also multiply by dp2 of all subtrees that hang off this maximal sequence. Note that the lengths of the paths in this maximal sequence are uniquely determined by their total length, and we need to check that the subtree sizes do not share any bits with each other and the length of this maximal sequence.

It remains to count the number of arrangements of paths that form this particular maximal sequence. Observe that the condition we must satisfy is that there are some $\ell = O(\log N)$ "checkpoints" that must be strictly crossed over by a path. We can count the number of arrangements using an inclusion-exclusion DP in $O(\ell^2)$ time.

The computation of $\mathrm{dp2}[u]$ is similar, except we enumerate one endpoint instead of two (unless the chain is trivial and $\mathrm{dp2}[u] = \mathrm{dp}[u]$).

In total, we process $O(\log^2 N)$ pairs of endpoints, each requiring $O(\log^2 N)$ time for the inclusion-exclusion DP. Therefore, the overall time complexity is $O(M + \log^4 N)$. The constant factors may be large (closer to $(4 \log_2 N)^4$), so some optimizations may be necessary to pass the time limit (such as not performing the inclusion-exclusion DP if intermediate results are already zero).

D Sell in Pairs

Problem Idea: Ditbul Ban
Problem Preparation: Ditbul Ban
Analysis By: Ditbul Ban

There are multiple solutions for this problem. We write the first-found solution here, which combines the approaches from Zixiang Zhou and Adhitya Mangudy Venkata Ganesh.

As the subtasks suggest, it is easy to tackle this problem by dividing cases to $X < Y$ and $X > Y$. Subtask 1, 2 corresponds to the $X < Y$ case, and subtask 3, 4 corresponds to the $X > Y$ case. We deliberately set the points of these two groups same to hide which one is easier, and encourage thinking in both directions.

It comes out that the $X > Y$ case is easier than $X < Y$ case. (We could call this a trap, if you started thinking $X > Y$ first because that subtask appeared earlier.) Subtask 5 handles some exceptional cases where $X = 0$, or $Y = 0$, or $X = Y$. The case $X = Y$ was excluded from the earlier subtasks since there were a few implementation methods which could be buggy if $X = Y$.

Subtask 3

In this subtask, $X > Y$ and $Q = 1$.

Let's define h_i and v_i as follows:

- h_i: the number of pairs $(i, i+1)$
- v_i: the number of pairs (i, i)

So we can think of minimizing $\sum h_i X + v_i Y$ where h_i and v_i should meet certain constraints. We will define possible pairing methods with these two sequences h and v, and for a method (or solution) $\mathcal{A}$, we will denote its profit as $p(\mathcal{A})$.

Observation D.1. *There is an optimal solution where $\forall i$, $h_i \leq 1$.*

Proof. Assume an optimal solution $\mathcal{A}$ contains some i where $h_i \geq 2$. Then, there are at least two pairs $(i, i+1)$. We can change them to pairs (i, i) and $(i+1, i+1)$, then we can decrease h_i by 2 (and alternatively adding 1 to v_i and v_{i+1}.) Let's denote this solution as $\mathcal{B}$.

Now, we show $\mathcal{A} < \mathcal{B}$. Note that everything changed from A to B is that A's two type 2 pairs changed to type 1 pairs. Therefore, $p(\mathcal{B}) - p(\mathcal{A}) = 2(X - Y) > 0$. Therefore, we can know that $\mathcal{A}$ is not an optimal solution, which contradicts the assumption. □

What does the above fact imply? This implies that for every i, there will be at most two mushrooms with flavor i which will be used in type 2 pairs. Therefore, if $A_i \geq 4$ for some i, we can freely pair two mushrooms with flavor index i to make a type 1 pair. This has an effect of decreasing A_i by 2, but since the resulting A_i will still satisfy $A_i \geq 2$, we know that choice was optimal.

Therefore, we can use this process to make $\forall i.A_i \leq 3$. From here, we can calculate the maximum profit for a single pair (X_i, Y_i) by using a simple linear DP in $O(n)$.

Subtask 4

Now our task is to quickly calculate the result of the above DP for multiple (X_i, Y_i) pairs.

First we will use an additional simple observation to make the situation more simple.

Observation D.2. *There is an optimal solution where for all i such that $A_i = 3$, $v_i = 1$.*

Proof. Assume there is some index i such that $A_i = 3$ and $v_i = 0$. Then,

- if $h_{i-1} + h_i \leq 1$, then we can add a type 1 pair in i, which does not worsen the solution.
- if $h_{i-1} + h_i \geq 2$, we can change one of those type 1 pairs to type 2 pair, which does not worsen the solution.

Therefore, we can always make an index i with $A_i = 3$ and $v_i = 0$ with $v_i = 1$. Repeating this process will give an optimal solution where $A_i = 3 \Rightarrow v_i = 1$. □

This property further allow us to limit $\forall i.A_i \leq 2$. Now we divide the group of i by the points where $A_i = 0$, and get the maximal regions $[l, r]$ where $\forall i \in [l, r].A_i \in \{1, 2\}$. Each regions are independent, so we can think region by region.

Here, one may think of the following approach:

- First, fill the $A_i = 2$ locations with type-1 pairs.
- Next, greedily fill the remaining two consecutive $A_i = 1$ locations with type-2 pairs, left to right.

While this method gets close to the optimal answer, there are countercases as follows: $N = 3$, $A = [1, 2, 1]$. The greedy algorithm above makes 1 type-1 pair which gets X points. However, when $X = 5$ and $Y = 4$, there is a way of making 2 type-2 pairs, which is better. Generally, we can observe the following facts.

- If a region contains $A_i = 2$ in the leftmost or rightmost area, it is optimal to fill them with type-1 pairs.

- Given two consecutive i-s with $A_i = 1$, consider the range between them (`122...221`). If there are k twos, the only two meaningful options are making k type-1 pairs or making $k + 1$ type-2 pairs.

These two facts are not difficult to prove, and in fact those two observations are heavily related to each other.

Therefore, our strategy is to first clean up the leftmost/rightmost $A_i = 2$ locations, and then for each `122...221` region, choose the optimal between using only type-1 pairs or type-2 pairs. Here, one thing to note is that we cannot use type-2 pairs for adjacent regions. These observations reduce the problem to the following one:

- For each region i, we could either choose profit $k_i X$ or $(k_i + 1)Y$.
- For two adjacent regions i and $i + 1$, we cannot choose both the Y option.
- What is the total maximum profit we can gain?

Let's choose, by default, the X option for all regions. Then, let's think about the change of profit by switching to the Y option. Then, we can modify the problem as follows:

- By modifying in region i, we get profit $(k_i + 1)Y - k_i X$.
- We cannot modify two adjacent regions.
- What is the total maximum profit we can gain?

This is a good form to solve by linear DP, but we want to solve this problem for multiple pairs of (X, Y). It is clear that we only need to think about the ratio of X and Y.

The problem above is related to a well-known problem of APIO 2007 Backup. Specifically, this problem asks the maximum profit when the number of region to modify is fixed. It is well known that this problem can be modeled as MCMF, and greedily adding augmenting paths is optimal. You should understand the solution of this problem first.

Now, in this problem, each augmenting path will add $m + 1$ modifications and remove m modifications. Since each modification has one more Y than a X, the profit gained from each modification will be in the form $(k + 1)Y - kX$. Here, note that

$$(k + 1)Y - kX = Y + k(Y - X) = (X - Y)\left(\frac{Y}{X - Y} - k\right)$$

If X and Y are fixed, the term $X - Y > 0$ and is fixed for all possible augmenting path, so we can ignore it for now and multiply it later. Let's define $\frac{Y}{X-Y}$ as r. Since the profit of each augmenting paths are nonincreasing, due to the convexity of MCMF, the optimal paths will have increasing value of k-s. Therefore, there will be a moment where a new augmenting path will actually give a negative profit, due to the value of r being smaller

than the k value of the next augmenting path. If we can precompute the augmenting paths, starting from small values of k, then we would be able to pinpoint the moment and calculate the exact maximum profit.

If you understood until this point, it would not be difficult to solve this problem, mainly using the similar idea with the APIO problem above, in $O((N+Q)\log N)$.

Subtask 1

In this subtask, $1 \leq X < Y$. In this case, answering for $Q = 1$ is much difficult than Subtask 3, since the main observations are much complicated.

We return again from the h_i, v_i approach. Now we know that there shouldn't be any two consecutive positive v_i values, since then we can change those two type-1 pairs to two type-2 pairs. Using this property, we can actually know a lot about the optimal solution. The main idea is starting from the left, and think about each individual location (flavor index), deciding how much type 1/2 pairs to make for the optimal solution. It turns out that even if we cannot decide it easily, we don't have many candidates.

We divide cases:

- If $A_1 \geq A_2 + 2$: it is optimal to make a type-1 pair $(1, 1)$. (If not, we can always make one, and it is always better.) Make enough type-1 pairs until $A_1 < A_2 + 2$, and continue to the cases below.
- If $A_1 = A_2 + 1$: here there are two possible cases which could be optimal. It is pretty straightforward that other options does not make sense.
  - Make one type-1 pair $(1, 1)$, and $A_2 - 1$ type-2 pairs $(1, 2)$.
  - Make A_2 type-2 pairs $(1, 2)$.
- If $A_1 \leq A_2$: it is optimal to make A_1 type-2 pairs $(1, 2)$. Assume that we spare some cells and make a type-1 pair $(1, 1)$. Then we can easily see that the two cells we spared in location 2 cannot be used for a (strictly) better way later on.

Therefore, we have at most two cases to diverge. After selecting how much type 1/2 pairs to make, we can proceed to the next index and continue. However, one might think that this divergions can be accumulated while we proceed from $i = 1$ to n, so we will have to take care of $O(n^2)$ cases in total. That is not true, since the only case of diverging here is when $A_1 = A_2 + 1$, where you get to make either A_2 or $A_2 - 1$ type-2 pairs. The adjacent cases, $A_1 = A_2$ or $A_1 = A_2 + 2$ all makes A_2 type-2 pairs, so there will be at most two cases with adjacent A_i value each time.

You can actually implement this using DP, and this solves each query in $O(N)$. Therefore, this solution solves Subtask 1.

Subtask 2

This is the most difficult subtask.

Writing the DP recurrence formula from Subtask 1, you can modify the formulas to be the following forms, excluding constants that would be added in every diverged path. We will write the formula in the sense that we are using memoization, starting from left and recursing through the right.

$$\begin{aligned}
(1)\quad & a_i = b_{i+1} + Y \\
& b_i = a_{i+1} \\
(2)\quad & a_i = \max(a_{i+2}, b_{i+2} + X) \\
& b_i = a_{i+2} \\
(3)\quad & a_i = a_{i+2} + X \\
& b_i = \max(a_{i+2}, b_{i+2} + X)
\end{aligned}$$

Note that in some cases we process two locations at once. Think about the two cases where $A_1 \geq A_2 + 1$ in the analysis of Subtask 1. In this case, most of the location 2 will be used with pairs, so we can limit the possibilities of $(2, 2)$ and $(2, 3)$ pairs, allowing us to compute two locations at one step.

From here we assume $Y = 1$, since only the ratio of X and Y is important. Note that (3) is same as (2) applied twice. Therefore, all we need is to process (1) and (2) well. We will reindex the terms since the index is not that important anymore.

$$\begin{aligned}
(1)\quad & a_i = b_{i+1} + 1 \\
& b_i = a_{i+1} \\
(2)\quad & a_i = \max(a_{i+1}, b_{i+1} + X) \\
& b_i = a_{i+1}
\end{aligned}$$

Since $b_i = a_{i+1}$ in all two cases, we can substitute it as follows:

$$\begin{aligned}
(1)\quad & a_i = a_{i+2} + 1 \\
(2)\quad & a_i = \max(a_{i+1}, a_{i+2} + X)
\end{aligned}$$

But by definition, $a_i \geq a_{i+1}$. Therefore, in (1), we can write $a_i = \max(a_{i+1}, a_{i+2} + 1)$. So, for each i, we have $a_i = \max(a_{i+1}, a_{i+2} + c_i)$, where $c_i \in \{1, X\}$, depending on i. Note that $X < 1 = Y$.

Now let's think about which values of X divide the final answer in a different form ($aX + b$ form). In this recurrence formula, the only way to gain value is to jump by two steps

at once. Therefore, the only way for two distinct paths to be not trivially comparable, without meeting in between, is only the case where one path takes $k+1$ X values and the other one takes k 1 value (or the same number of X or 1 are added to both paths.) This shows that, if we draw a graph with X in the x-axis and the optimal answer in the y-axis, the point of changes are only where X is in the form of $\frac{k}{k+1}$ for some $k \in \mathbb{Z}_+$, where $1 \leq k \leq$ (the size of interval of interest).

Therefore, there will be at most $O(n)$ line segments in the graph. From here, you can maintain this convex graph of line segments, using divide and conquer, to maintain the answer for $0 < X/Y < 1$. The solution will run in $O((n+q)\log n)$.

Subtask 5

This subtask covers these three cases:

- $X = 0$
- $Y = 0$
- $X = Y$

These cases are separated from the earlier subtasks since some implementions of the earlier subtasks could be wrong in these special cases. $X = 0$ and $Y = 0$ are easily solvable by greedy algorithms, and $X = Y$ can be done with the Subtask 4 solution above.

E Planar Exact Cover

Problem Idea: Zixiang Zhou
Problem Preparation: Zixiang Zhou
Analysis By: Zixiang Zhou

This problem is solved in the paper `https://arxiv.org/abs/2603.03488` (see Section 3.2.1). We summarize the key ideas and discuss a near-linear time implementation here.

> MIT Hardness Group: Zachary Abel, Erik D. Demaine, Jenny Diomidova, Jeffery Li, and Zixiang Zhou. Planar Graph Orientation Frameworks, Applied to KPlumber and Polyomino Tiling. arXiv:2603.03488, 2026.

Let $k = 6$.

Intuitively, planarity combined with the large degree k significantly constrains the graph, to the point where we expect there to be local structure that allows us to "simplify" the problem instance. For example, a preliminary observation is that there must be a stall of degree at most 2:

Lemma E.1. *There does not exist a simple, nonempty, planar bipartite graph where all vertices on one side of the bipartition have degree k and all vertices on the other side have degree at least 3.*

Proof. Let x be the number of vertices on the side with degree k vertices, and let y be the number of vertices on the other side. There are kx edges, so

$$kx = \sum \deg(v) \geq 3y \implies y \leq \frac{kx}{3}.$$

We also have the bound $|E| \leq 2|V| - 4$ for nonempty planar bipartite graphs, so

$$kx \leq 2(x+y) - 4 \leq 2\left(x + \frac{kx}{3}\right) - 4 \implies \frac{kx}{3} \leq 2x - 4.$$

This is a contradiction for $k = 6$. □

Any stall of degree 1 forces its unique adjacent barn to be selected, and any stall of degree 2 forces exactly one of its adjacent barns to be selected, and we can imagine somehow "merging" these barns together into one object.

To describe details more concretely, let us reformulate the problem. Consider orienting each edge so that every selected barn has all k of its edges pointing outward and every unselected barn has all k of its edges pointing inward. We require that every stall has exactly 1 incoming edge and every barn has 0 or k incoming edges (i.e., the edges have "equal" orientation). Thus, the problem is equivalent to the following **Graph Orientation** problem:

> Given a planar graph where vertices are labeled either 1-*in*-j (where j is its degree) or k-*equalizer* (where it must have degree k), count the number of ways to orient the edges to satisfy the constraints at each vertex.

We claim that the number of solutions is either 0 or a power of two, with the corresponding solution set being *affine* (some edges have a forced orientation, some pairs of edges are forced to have equal orientations, or forced to have opposite orientations).

In the Graph Orientation formulation, we repeatedly apply local reductions to simplify the problem instance. These reductions include contracting 1-in-2 vertices into a single edge, merging two adjacent 1-in-j_1 and 1-in-j_2 vertices into a 1-in-$(j_1 + j_2 - 2)$ vertex, and eliminating self-loops and multi-edges created during the course of the algorithm. The trickiest reduction is combining two adjacent k-equalizer vertices (which comes from two barns sharing a degree-2 stall). We handle it as follows:

> If there is at least one edge (possibly multiple edges) between two distinct k-equalizer vertices, say u and v, take one such edge uv. Starting from this edge, let v followed by $u_1, \ldots, u_{k'}$ be the neighbors of u other than v in clockwise order, and similarly let u followed by $v_1, \ldots, v_{k'}$ be the neighbors of v other than u in counterclockwise order ($0 \le k' < k$). Form a new graph by deleting u and v and connecting each u_i with v_i, which preserves planarity. We temporarily "throw away" the constraints that all the $u_i v_i$ edges should point in the "same" direction and solve the resulting graph, which has an affine solution set. Then, add back these constraints to get the solution set of the original graph.

If implemented correctly, we only reach a situation with no more applicable reductions when the graph is simple and bipartite with 1-in-j and k-equalizer vertices forming a bipartition, and all 1-in-j vertices have $j \ge 3$. From Lemma E.1, the graph must be empty at this time and we are done.

With a careful implementation (a queue of applicable reductions, small-to-large merging of adjacency lists), this algorithm can be implemented in $O((N+M)\log(N+M))$ time.

Remark. The same algorithm works for any $k \ge 6$. With one added reduction rule, the problem is still solvable in polynomial time if $k = 5$. However, $k = 3$ or $k = 4$ are NP-complete.