Problem Analysis

Disclaimer: *This is an analysis of some possible ways to solve the problems of The 2019 ICPC Asia Jakarta Regional Contest. Since the purpose of this analysis is mainly to give the general idea to solve each problem, we left several (implementation) details in the discussion for reader's exercise. If some of the terminology or algorithms mentioned below are not familiar to you, your favorite search engine should be able to help.*

Problem Title		Problem Author	Analysis Author
A	Copying Homework	Jonathan Irvin G., Suhendry E.	Suhendry Effendy
B	Cleaning Robots	Suhendry Effendy	Suhendry Effendy
C	Even Path	Vincentius Madya Putra Indramawan	Suhendry Effendy
D	Find String in a Grid	Wiwit Rifa'i	Wiwit Rifa'i
E	Songwriter	Alfonsus Raditya Arsadjaja	Alfonsus Raditya Arsadjaja
F	Regular Forestation	Winardi Kurniawan	Suhendry Effendy
G	Performance Review	Jonathan Mulyawan Woenardi	Jonathan Mulyawan Woenardi
H	Twin Buildings	Wiwit Rifa'i	Wiwit Rifa'i
I	Mission Possible	Vincentius Madya Putra Indramawan	Suhendry Effendy
J	Tiling Terrace	Ashar Fuadi	Ashar Fuadi
K	Addition Robot	Prabowo Djonatan	Prabowo Djonatan
L	Road Construction	Jonathan Irvin Gunawan	Jonathan Irvin Gunawan

A. Copying Homework

There are multiple ways to get CORRECT in this problem. To make sure the output B has $\geq N$ total difference with A, we may need to shuffle A such that no element in A remains in its position (also called a *derangement* of A), i.e. $A_i \neq B_i$ for all $i = 1..N$. One easy method to achieve this is simply by shifting A one element to the right (or left). For example, let $A = \{1, 2, 3, 4, 5\}$, then $B = \{2, 3, 4, 5, 1\}$. Using this method, it is guaranteed that no element in A remains in its position.

Several WRONG-ANSWER solutions:

- Reversing A. When $|A|$ is odd, then reversing A will leave exactly one element (at the center) remains in its position. It is possible to design cases such that the total difference is less than N. For example, $A = \{2, 4, 3, 5, 1\}$ and $B = reverse(A) = \{1, 5, 3, 4, 2\}$, then the total difference is $1 + 1 + 0 + 1 + 1 = 4$ which is less than $N = 5$.
- Random shuffling A. It is tempting to simply random shuffle the input to solve this problem, however, such a solution would not work. There is no guarantee that random shuffling A will produce B such that $A_i \neq B_i$ for all i, thus, the total difference might be less than N. The test data is designed such that a **plain** random shuffle (e.g., C++ `random_shuffle()`, Java `Collections.shuffle()`, Python `random.shuffle()`) solution only has $< 0.02\%$ chance of getting CORRECT. Moreover, if a fixed random seed is used, then it has 0% chance of getting CORRECT.

B. Cleaning Robots

This problem can be solved with dynamic programming. The following analysis assumes the tree is rooted (simply select an arbitrary node as the root).

First, let us define a state $\langle u, s\rangle$ where u is a node and s is either $\{0, 1, 2\}$ as follows:

- $\langle u, 0\rangle$ – node u and its parent are cleaned by a different robot.
- $\langle u, 1\rangle$ – node u and its parent are cleaned by a different robot, and node u should be cleaned together with its 2 direct children.
- $\langle u, 2\rangle$ – node u and its parent are cleaned by the same robot.

Let a function $f(u, s)$ be the number of feasible deployment plans on a subtree rooted at node u. Then, the solution to this problem is simply $f(root, 0)$.

With these state and function definitions, we can derive the recurrence relation between states. Let η^u be the set of node u's children, η^u_i be the ith child of node u, and n be the number of node u's children (i.e. $|\eta^u|$). The recurrence relation is:

$$f(u, s) = \begin{cases} h_1(u, 1) + h_2(u) + h_3(u) & \text{if } s = 0, \\ h_2(u) & \text{if } s = 1, \\ h_1(u, 0) + h_3(u) & \text{if } s = 2 \end{cases}$$

$$h_1(u,s) = \sum_{i=[1..n]} \prod_{v \in \eta_u} \begin{cases} f(v,2) & \text{if } v = \eta_i^u \\ f(v,s) & \text{if } v \neq \eta_i^u \end{cases}$$

$$h_2(u) = \sum_{i=[1..n]} \sum_{j=(i..n]} \prod_{v \in \eta_u} \begin{cases} f(v,2) & \text{if } v = \eta_i^u \text{ or } v = \eta_j^u \\ f(v,0) & \text{if } v \neq \eta_i^u \text{ and } v \neq \eta_j^u \end{cases}$$

$$h_3(u) = \prod_{v \in \eta_u} f(v,1)$$

The function $h_1(u,s)$ handles the case where we should choose one child to be cleaned together with node u while the remaining other children are in state $\langle \eta_i^u, s \rangle$. The function $h_2(u)$ handles the case where we should choose two children to be cleaned together with node u while the remaining other children are in state $\langle \eta_i^u, 0 \rangle$. The function $h_3(u)$ handles the case where node u is not cleaned together with any of its children.

Observe that the time complexity to compute $h_1(u,s)$ naïvely is $O(n^2)$, while $h_2(u)$ is $O(n^3)$, and $h_3(u)$ is $O(n)$. Therefore, the time complexity to compute the above recurrence relation is $O(N^3)$, which is too slow for this problem with $N \leq 100\,000$. Optimizing these functions is the real challenge for this problem.

First good news: The function $h_1(u,s)$ can be computed in $O(n)$.

$$h_1(u,s) = \prod_{v \in \eta_u} f(v,s) \times \sum_{v \in \eta_u} f(v,2) f^{-1}(v,s)$$

Note that we need to use the multiplicative modular inverse to compute $\frac{1}{f(v,s)}$ as we only need to find the modulo while the full number can be very large.

Another good news: The function $h_2(u)$ can be computed in $O(n)$.

$$x_i^u = f(\eta_i^u, 2) f^{-1}(\eta_i^u, 0)$$

$$y_i^u = \sum_{j=[i..n]} x_j^u$$

$$h_2(u) = \prod_{v \in \eta_u} f(v,0) \times \sum_{i=[1..n]} x_i^u y_{i+1}^u$$

Note that y_i^u can be computed in $O(n)$ for all i with partial sum technique, making it $O(1)$ for each i. Also note that $y_i^u = 0$ if $i > n$. The original $h_2(u)$ function is actually in the form of $a_1a_2b_3b_4b_5 + a_1b_2a_3b_4b_5 + a_1b_2b_3a_4b_5 + \cdots + b_1b_2a_3b_4a_5 + b_1b_2b_3a_4a_5$ (an example where $n = 5$), i.e. there are 2 functions, a and b, choose two indexes to have a while the remainings are b; evaluate each combination (perform the multiplications) and sum the results for all the possible combinations. There are $\binom{n}{2}$ combinations, and evaluating one combination is $O(n)$, thus, the original time complexity is $O(n^3)$. Try to optimize this summation into $O(n)$ and you will arrive at the above equations.

We do not need to optimize the function $h_3(u)$ as it is already in $O(n)$. All seems good! The solution now becomes $O(N)$. However, there is a caveat in the implementation: Beware when you want to compute the modular multiplicative inverse of zero (there is no such thing). You'll need to further work on the equations on such a case. Hint: We only need to consider the cases when there is at most 1 zero for $h_1(u)$ and 2 zeros for $h_2(u)$.

C. Even Path

Observe that we can go from cell (r, c) to cell $(r \pm 1, c)$ only if the parity of $R[r]$ and the target cell (i.e. $R[r \pm 1]$) are the same because there is only one element changing, i.e. $R[r]$ to $R[r \pm 1]$. Similar things also happened with the column. We can go from cell (r, c) to cell $(r, c \pm 1)$ only if the parity of $C[c]$ and the target cell (i.e. $C[c \pm 1]$) are the same.

Therefore, for each query $\langle r_a, c_a, r_b, c_b \rangle$, we only need to check whether the parity of $R[r]$ are the same for all $r = \min(r_a, r_b)..\max(r_a, r_b)$, and whether the parity of $C[c]$ are the same for all $c = \min(c_a, c_b)..\max(c_a, c_b)$. Do a precomputation first (in $O(N)$) before processing any query so that we can decide whether $R[i..j]$ or $C[i..j]$ have the same parity in $O(1)$ for all pair of i and j. You can use a union-find data structure to do this, although it is overkill; there is another much simpler ad-hoc method to find the groups involving only one iteration. The time complexity for this solution is $O(N + Q)$.

D. Find String in a Grid

To count the number of occurrences of the string S_i in the grid, we can try each character position on S_i as a turning point. So, the total number of tries are $\sum_{i=1}^{Q} |S_i| \leq 200000$. Let us choose j^{th} character position as the turning point, then the string S_i is consisted of a horizontal string H and a vertical string V. The string H is $S_{i1}S_{i2} \ldots S_{ij}$ and the string V is $S_{ij}S_{ij+1} \ldots S_{i|S_i|}$. To make the explanation become easier, we will reverse the string H as $S_{ij}S_{ij-1} \ldots S_{i1}$.

For case $|H| = 1$, we can ignore H and count the number of occurrences of string V vertically with a suffix array. The suffix array will sort every cell (r, c) based on the suffix string $G_{r,c}G_{r+1,c} \ldots G_{R,c}$. So, every vertical occurrences of string V on the grid will be adjacent each other in the suffix array, so we can count the number of occurrences of string V based on the longest common prefix of every adjacent indices on suffix array and use binary search with the help of data structure such as range minimum query to find the range of the occurrences of string V. We can insert all query strings to the suffix array before so that we can know the starting position to do the binary search.

The equivalent process is also applicable for case $|V| = 1$, i.e. we can find the number of occurrences of string H horizontally with suffix array by sorting every cell (r, c) based on the suffix string $G_{r,c}G_{r,c-1} \ldots G_{r,1}$.

If $|V| \neq 1$ and $|H| \neq 1$, then we can combine those 2 suffix arrays. Suppose the position of cell (r, c) on the first suffix array (based on vertical suffix) is $X_{r,c}$ and the position of cell (r, c) on the second suffix array (based on horizontal suffix) is $Y_{r,c}$. Since every occurrences of string V are adjacent on first suffix array (e.g. it starts from L_V until R_V) and every occurrences of string H are adjacent on second suffix array (e.g. it starts from L_H until R_H), then to combine V and H become together, we need to count how many cell (r, c) that satisfying $L_V \leq X_{r,c} \leq R_V$ and $L_H \leq Y_{r,c} \leq R_H$.

We can model the rest problem as 2-D Cartesian coordinates. Each cell will be modeled as a point $(X_{r,c}, Y_{r,c})$

and each try on choosing the turning point will become a rectangle from point (L_V, L_H) until point (R_V, R_H). So, the rest problem is to count how many points inside each rectangle. This problem can be solved by data structure such as range tree or we can also use line sweep technique and use range sum query data structure such as BIT or segment tree.

The time-complexity of this solution is $O((R \times C + \sum_{i=1}^{Q} |S_i|) \times \log(R \times C + \sum_{i=1}^{Q} |S_i|))$.

E. Songwriter

First of all, we can ignore the original sequence A and just get the relation between every 2 consecutive numbers in A. At first glance, it looks like it can be done with a greedy approach from left to right; but if the sequence is almost all decreasing or most of them are decreasing, we have to make an adjustment to the elements prior to the current element when making B. It can be seen that the worst case is for every position, we will be updating all numbers below the current position, results in $O(n^2)$. We need something faster.

It can be seen that for every position, the numbers that have solutions formed a range. We define 2 functions, low and $high$, that receives i as an input and returns the lowest and highest number that have a solution for position i, respectively; So for every position i $(1 \leq i \leq N)$, every number x in which $low_i \leq x \leq high_i$, we can find a solution that contains x as a number in position i of the new sequence B.

Note: If two consecutive elements have equal value, the values of 2 functions in corresponding indices will be equal too. So for simplicity, we ignore the consecutive elements that have equal value and only consider the rest.

We can generate the values of low and $high$ as the following:

$$(low_i, high_i) = \begin{cases} (L, R), & i = N \\ (\max(low_{i+1} - K, L), high_{i+1} - 1), & 1 \leq i < N, A_i < A_{i+1} \\ (low_{i+1} + 1, \min(high_{i+1} + K, R)), & 1 \leq i < N, A_i > A_{i+1} \end{cases}$$

If there exists some position i that $low_i > R$ or $high_i < L$, there is no solution. Otherwise, there exists a solution for given constraints and configuration. As we want the lexicographically smallest sequence, we generate B from the smallest position.

The element that fills the first position (position 1) will be low_1 itself, as it is the lowest possible number that has a solution. In every following position, we can simulate the similar algorithm with the low and $high$ function above, but this time only considers the previous number that just been generated, and the low and $high$ functions that have been generated before. So if $A_{i-1} < A_i$, B_i must be somewhere inside $[B_{i-1} + 1, B_{i-1} + K]$, and vice versa. But B_i must also somewhere inside $[low_i, high_i]$, so we just check the lowest number that falls into both range, and it gives the value of B_i. It is guaranteed that there exists a number that falls into both ranges at the same time, as we already checked there is a solution first.

This solution runs in linear time, hence $O(N)$, enough to pass within the time limit.

F. Regular Forestation

The *centroid* of a tree is a node whose removal causes the remaining tree(s) to have at most half the number of nodes of the original tree. A tree can only have at most 2 centroids.

Recall that removal of a good cutting point (as defined in the problem statement) will cause at least two identical disconnected trees. Observe that each of those disconnected trees cannot have more than half the number of nodes of the original tree. Therefore, if a good cutting point exists in the given tree, then it must be at the centroid of the tree.

Tree centroid can be found in $O(N)$–alternatively, an $O(N \log N)$ or $O(N^2)$ if implemented efficiently, can also be used as N in this problem is quite small.

The remaining task is to check whether the disconnected trees are identical (also called *isomorphic*). Rooted tree isomorphism can be solved in $O(N)$ with parenthetical tuples (AHU algorithm), although an easier $O(N^2)$ version suffices. For this problem where the trees are unrooted, simply use the centroid of each tree as the root. You need to be careful when the tree has 2 centroids. If you opt to test all nodes in each tree as the root for tree isomorphism, then you will need the $O(N)$ algorithm to do the tree isomorphism.

Note that using the *center* of a tree (as opposed to centroid) as the only candidate for the good cutting point will not work (WRONG-ANSWER). The center of a tree is a node that has the lowest eccentricity (or lies in the diameter path of a tree).

G. Performance Review

We first observe that, if there is an employee with better performance than Randall gets replaced in a particular year, then Randall must have also been replaced at the same time or before the particular employee is replaced. Therefore, for Randall to stay in the company after M years, all employees (original and newly added) who have better performance than Randall must also stay in the company after M years. We only need to check for each year, whether the number of employees with better performance than Randall plus Randall itself is less than or equal to the number of employees not replaced on that particular year.

Let $R[i]$ be the number of employees not replaced subtracted by the number of employees with better performance than Randall plus Randall itself on that particular year. Construct the initial array and check whether the minimum value of $R[i]$ for all i is greater than or equal to 0.

For each query, there are 4 possible cases

1. The employee is previously worse than Randall. After the change, the employee is still worse than Randall.
2. The employee is previously worse than Randall. After the change, the employee is now better than Randall.
3. The employee is previously better than Randall. After the change, the employee is now worse than Randall.
4. The employee is previously better than Randall. After the change, the employee is still better than Randall.

For case 1 and case 4, since there is no change in the number of employees better than Randall joining the company each year, the answer stays the same as before the query happens. For case 2, the number of employees in the company better than Randall increased by 1 starting from the next year after the change until the last year. For case 3, the number of employees in the company better tan Randall decreased by 1 starting from the next year after the change until the last year.

We can use Segment Tree with Lazy Propagation to support our update and query requirement. The query type is range minimum query and the update type is range update of +1 or -1.

The time complexity will be $O(N + (M + Q) \log M)$.

H. Twin Buildings

For each land, we can always build the two buildings on the same land by dividing this land into half horizontally or vertically. So, the answer should be at least half of the largest area of all buildings.

Since the two buildings don't necessarily have the same orientation if they are on the different lands, then we can assume that we can always swap L_i and W_i for any i^{th} land. If we place the two building on i^{th} and j^{th} land, then the largest building area that we can build is $\max(\min(L_i, L_j) \times \min(W_i, W_j), \min(L_i, W_j) \times \min(W_i, L_j))$.

Without loss of generality, we assume that $\min(L_i, W_i, L_j, W_j) = L_i$, then the largest building area that we can build on i^{th} and j^{th} land is $\max(L_i \times \min(W_i, W_j), L_i \times \min(W_i, L_j))$ or become $L_i \times \min(W_i, \max(W_j, L_j))$. To make the solution easier, we can swap L_i and W_i for some lands so that $L_i \leq W_i$ for all buildings.

Let us sort the lands based on the non-increasing L_i. Observe that for every $i > j$ we have $L_i \leq L_j$, and then the largest building area that we can build is $L_i \times \min(W_i, \max(W_j, L_j)) = L_i \times \min(W_i, W_j)$, since $L_j \leq W_j$. So, for any i, we just need to find the largest W_j for all $j < i$. We can do all of them in $O(N \log N)$ time-complexity.

Note: we must avoid using floating-point numbers since the answer can be very large until 10^{18} and it might cause the precision loss.

I. Mission Possible

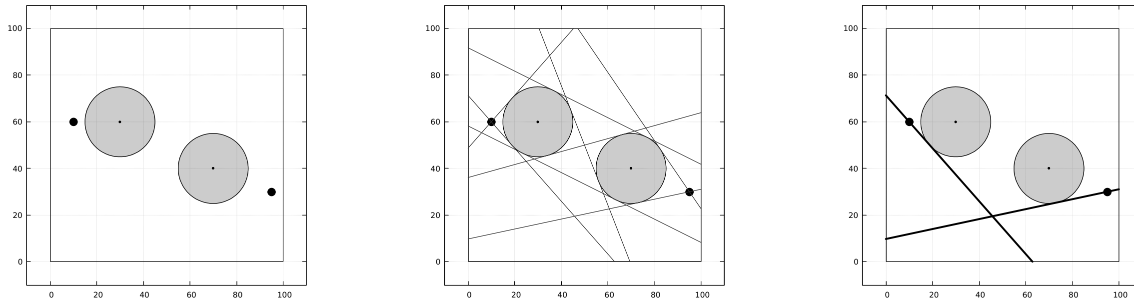
We need a basic understanding of graph theory (pathfinding) and geometry (tangent and line) to solve this problem. The big idea is: The feasible path can be found through some tangent/boundary line segments.

First, let us deal with the geometry aspect of this problem. Find the tangent lines between Allen's initial/target points to any circles, and the tangent lines between any two circles. Next, make each tangent line into a tangent line segment such that:

1. The two objects (points or circles) which create the tangent line are still connected by the line segment,
2. The line segment spans as much as possible without crossing any circle or going out of the boundaries.

If the tangent line segment cannot connect the two objects which create it (e.g., it crosses another circle), then we can drop that tangent line. Next, find all the boundary line segments; there are 4 boundaries (the

rectangle's sides), but we need to break them down into some line segments if there is a circle crossing the boundaries.



Now, let us transform this problem into a graph problem. Let the line segments be the vertices, and an edge implies that the two line segments have an intersection point. The starting vertex is all line segments that go through Allen's initial point, while the goal vertex is all line segments that go through Allen's target point. Finally, simply do a *breadth-first search* (BFS) to find the path from the starting vertex to the goal vertex. You need to keep the parent node as you do BFS to print the points.

J. Tiling Terrace

The solution uses a combination of dynamic programming and greedy. Let S be the number of characters '.' (soils) in the input, and R be the number of characters '#' (rocks) in the input. Note that R is at most 50.

Observation 1 Since type-3 tiles can only be placed on ".#.", then the maximum possible number of type-3 tiles that can be placed is R (which is at most 50).

Observation 2 Because we want to maximize the result, any placed type-2 tile can be replaced by two type-1 tiles (when $G_1 * 2 > G_2$), or one type-1 tile (when $G_1 > G_2$ and doing the former would exceed the limit of K).

The solution uses dynamic programming by assuming that we can only use type-2 and type-3 tiles. Type-1 tiles can then be inserted greedily using Observation 2.

Define $dp[i][j]$ be maximum number of type-2 tiles that can be used for tiling cells $[1..i]$, using exactly j type-3 tiles. The recurrence is trivial: For each state, decide whether we want to leave the ith cell untiled, put type-2 tile, or put type-3 tile.

After we fill our DP table, we can brute-force j, the number of type-3 tiles used. For each j, we know that:

- The number of type-2 tiles used is $dp[N][j]$
- The number of type-3 tiles used is j
- The number of untiled soil cells is $S - 2 * dp[N][j] - 2 * j$
- We can put type-1 tiles on the untiled cells. Then, if we still have spare type-1 tiles left, we can greedily replace some of the type-2 tiles, using Observation 2.

The answer is the maximum result over all j. The time complexity is $O(NR)$.

K. Addition Robot

Let the transformation matrix for characters 'A' and 'B' be $\begin{bmatrix}1 & 0\\1 & 1\end{bmatrix}$ and $\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}$ respectively. This is because:

$$\begin{bmatrix}a & b\end{bmatrix}\begin{bmatrix}1 & 0\\1 & 1\end{bmatrix} = \begin{bmatrix}a+b & b\end{bmatrix} \text{ and } \begin{bmatrix}a & b\end{bmatrix}\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix} = \begin{bmatrix}a & a+b\end{bmatrix}$$

Next, we build a segment tree of size N, where each node is a 2×2 matrix which is the multiplication of its two children. The i-th leaf node is $\begin{bmatrix}1 & 0\\1 & 1\end{bmatrix}$ if $S[i] = A$ or $\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}$ if $S[i] = B$.

To answer the query, we simply do the standard range query operation on the segment tree and obtain the transformation matrix. Multiply the initial A and B with this matrix will give us the answer.

To serve the toggle operation, we do update on the segment tree (with lazy propagation). If a node containing $\begin{bmatrix}p & q\\r & s\end{bmatrix}$ covers the range that is to be updated, we update the node to $\begin{bmatrix}s & r\\q & p\end{bmatrix}$.

We can prove the correctness of the toggle operation by induction on the length of the character. The base case of a single character can obviously be seen as true. Suppose the toggle operation is true for a string of length k. Let $\begin{bmatrix}p & q\\r & s\end{bmatrix}$ be the transformation matrix of the string of length k. Adding the character A will give the transformation matrix:

$$\begin{bmatrix}p & q\\r & s\end{bmatrix}\begin{bmatrix}1 & 0\\1 & 1\end{bmatrix} = \begin{bmatrix}p+q & q\\r+s & s\end{bmatrix}$$

Applying the toggle operation on the left hand side gives us:

$$\begin{bmatrix}s & r\\q & p\end{bmatrix}\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix} = \begin{bmatrix}s & r+s\\q & p+q\end{bmatrix}$$

The prove for "adding character B" can be done similarly. Hence, the toggle operation is proven to be true.

The total complexity is $O(Q \log N)$.

L. Road Construction

Let us consider the easier version of this problem first if there are only $N-1$ (instead of N) road proposals. In this case, all roads have to be constructed. Therefore, we need to find a surjective mapping from the set of road proposals to the set of workers.

We can do this by modeling a bipartite graph where we created a node for each road proposal and worker. We add an edge connecting a road proposal node and a worker node if the worker can construct the road proposal. The maximum cardinality bipartite matching should give us a surjective matching from the set of road proposals to the set of workers if it exists. We will have $O(NK)$ edges with this approach.

We can reduce the number of edges by creating a node for each material (that exists in the input) instead of for each worker. Each of these nodes has a capacity (the number of times this node can be matched) equals to the number of workers that is familiar with the material. We add an edge connecting a road proposal node and a material node if the road proposal can be constructed with the material. Since a node u will be connected to M_u nodes, we will have $O(\Sigma_{i=1}^{N} M_i)$ edges with this approach.

Now let's get back to the original problem where we have N edges. Therefore, the road proposals will contain exactly one cycle. Let's denote A as the set of road proposals inside the cycle and B as the set of road proposals outside the cycle. We need to construct at least $|A| - 1$ roads in set A and all roads in set B.

There are two ways to do this. Since we want to avoid redundant worker assignment, we want to keep at most $|A| - 1$ workers to construct the set of road proposals in A. Therefore, we can create an additional node (with a capacity of $|A| - 1$) that is connected to the maximum flow source and the set of road proposal nodes in A. Alternatively, the easier method is to run the maximum cardinality bipartite matching on set B first, before running it on set A. We will still have $O(\Sigma_{i=1}^{N} M_i)$ edges with this approach.