OPTIMAL POLYGON CONSTRUCTION

Team 57

Understanding The Problem Statement...

KRITI

01

Input:

Number of the stellar crystals, position of each stellar and value , followed by number of void mines , position of each mine and its risk

02

Constraints:

Total mines and stellars are at most 10000, space is 10000*10000 and edges of polygon are parallel to x and y axis having at most 1000 vertices.

03

Goal:

Maximum net value inside a polygon : (total value of crystals) - (total risk of mines).

Roadmap to optimization...

KADANE

By dividing the columns in group of 40 we will have 250 columns we will find maximum subarray and connect them. this approach gives average **54.2%** of total positive sum

DOUBLE KADANE

By slightly modifying Kadane's approach, we maintain the top two maximum subarrays and select the top 200 boxes, then connect them. this approach gives average **55.8%** of total positive sum

DP

In the DP approach, we determine whether to take 0, 1, or 2 subarrays for a given column while also tracking the number of vertices used at any instance.

VARIABLE WIDTH

We adjust the column widths to 1, 40, and 50, selecting the one that gives the maximum value. The widths 40 and 50 are chosen experimentally, while 1 handles edge cases. We also consider the transpose of the matrix.

EDGES

By tracing the DP table backward, we identify the edges of the polygon by usingthe choices made during the computation. This helps in reconstructing the selected subarrays.

Being greedy: choosing from a column

Creating Divisions

The entire Grid is divided into columns of a meticulously chosen width of 40.
Considering this amalgamation of 40 columns as a subgrid, we are left with 10000/40 many subgrids which are names as Columns from now.

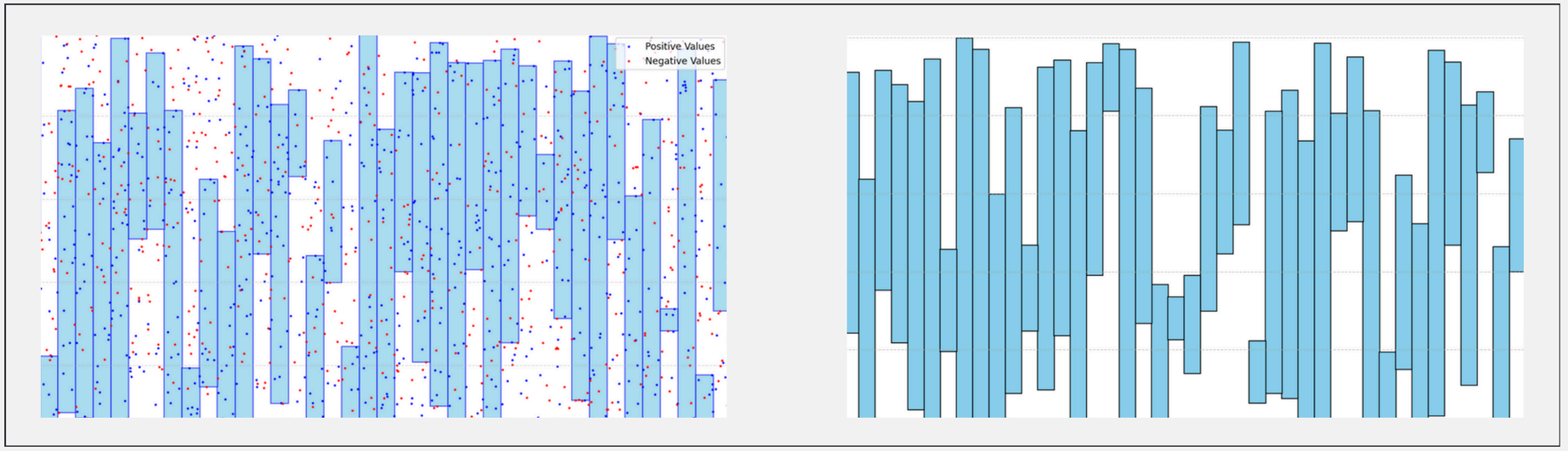
Using Kadane's Algorithm

Now we need to choose some sub-array out of every grid which would then be connected via extra edges forming polygons.

We here choose this Greedily by finding the Maximum subarray sum of each Column.

Can we optimize further?

Using this Naive Approach, average accuracy of ans when calculated on given test-cases (out of the total positive available) was ***54.2%***.
What if the Second Maximum sub-array from each column is also chosen?



Boxes Representing Maximum Subarray Sums.

Being greedier: second maximum sum

Using Kadane's Two Times

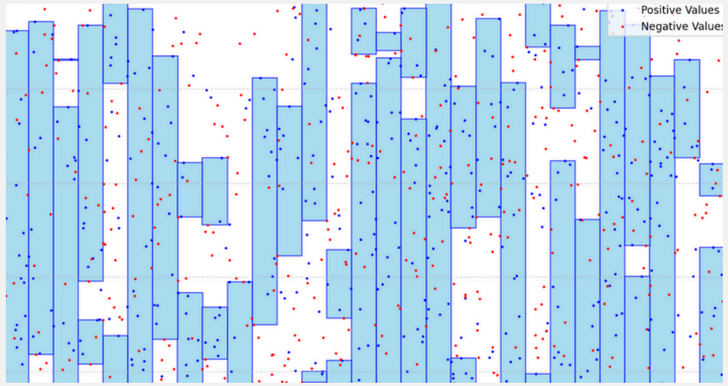
After finding the maximum sub-array sum of all the columns, we made those values negative infinity to calculate the second maximum sub-array sum by applying **Kadane's** again on the columns.

Can It Be Used Third Time?

Through extensive testing, we observed that the third maximum sub-array sum was negligible compared to the first two and also increased the number of boxes and hence vertices. Therefore, we optimized our approach by considering only the **first two** maximum subarrays.

Enough Being Greedy . . .

Using this Optimized Approach and selecting **top 200** boxes (to avoid exceeding vertices) according to their sums, average accuracy of answer (out of the total positive available) turned out to be ***55.8%***.
Could there be a more optimal **dynamic** solution to avoid the chaos of number of vertices?



Boxes Representing Two Maximum Subarray Sums.

Which one to choose...Dynamic Programming

KRITI

Why DP?

Follows from the basic Kadane's algorithm presented, on the top of which Dynamic Programming's layer is required to optimally choose between Single-Double sub-arrays of the columns where Greedy would fail.
Max Subarrays: Identified the two maximum sum sub-arrays per division using Kadane's algorithm.

Segment Width

To optimize segmentation, the total range of 10,001 units is divided into 250 segments. The first segment has a width of 41 units, while the remaining 249 segments are uniformly distributed with a width of **40 units** each. This structured approach ensures balanced distribution while maintaining precision in partitioning.

DP states: dp[i][j][k]

i = {0 , 1 , 2} j = {0 ... 249} k= {0 ... 1000}

k: No of vertices used.

j : Index of the current division.

i : 0 = No subarrays chosen in current division.
1 = Only first maximum subarray chosen in current division.
2 = Both first and second maximum subarrays;

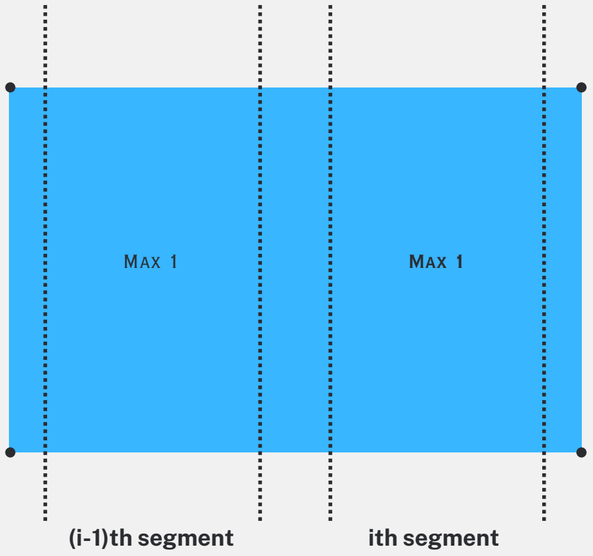
What the DP means?

DP[i][j][k] represents the optimal value that can be achieved when processing the first j divisions while using exactly k vertices, i is the number of maximum subarrays selected in the current division, where i = 0 means no subarrays are chosen, i = 1 means only the first maximum subarray is chosen, and i = 2 means both the first and second are chosen.

How many points?

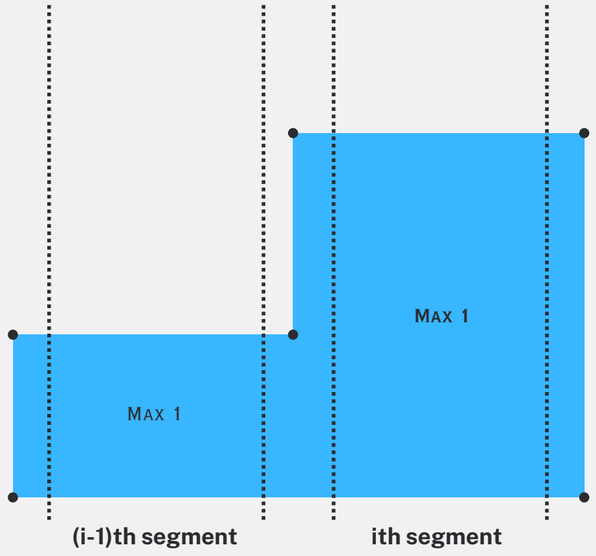
0 Vertices
Both matches

If both the starts and ends of the i^{th} and $(i-1)^{th}$ subarrays match, then no extra vertices are required for the new subarray.



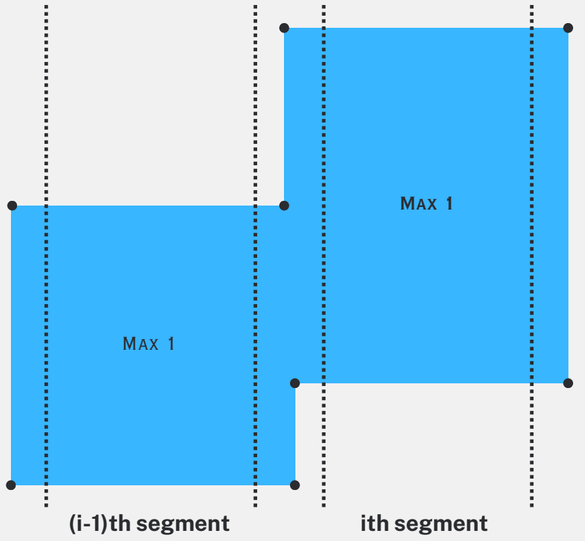
2 Vertices
One matches

If either only the start or only the end of the i^{th} and $(i-1)^{th}$ subarrays match, then they would require 2 extra vertices for them to be joined.



4 Vertices
Nothing matches

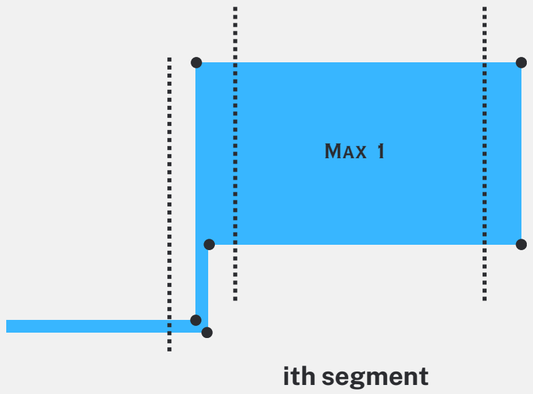
If the i^{th} and $(i-1)^{th}$ subarrays do not match in any way, constructing the polygon would require exactly 4 extra vertices to ensure proper connectivity and structure.



How many vertices?

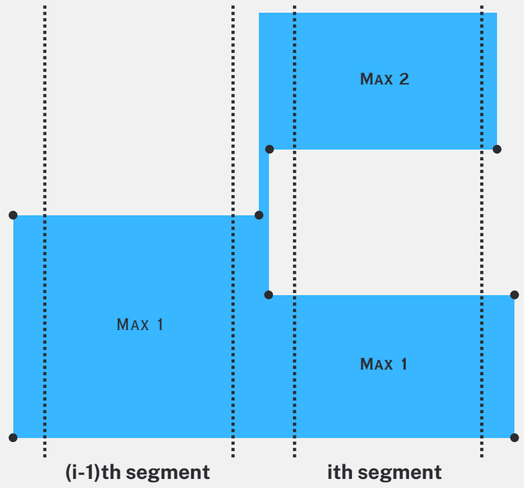
6 Vertices
$(i-1)^{th}$ not taken

If the $(i-1)^{th}$ subarray is not taken, but the i^{th} subarray is taken, then we need 6 extra vertices (in the worst case) - 2 of them for the bend in pipeline.



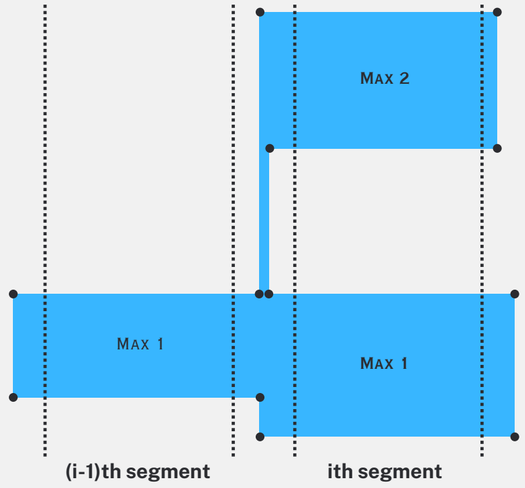
6 Vertices
Bottom matches

In the case of both subarrays chosen in i^{th} division, if both start and end are same as the $(i-1)^{th}$ division, then only 6 extra vertices will be required.



8 Vertices
Top matches

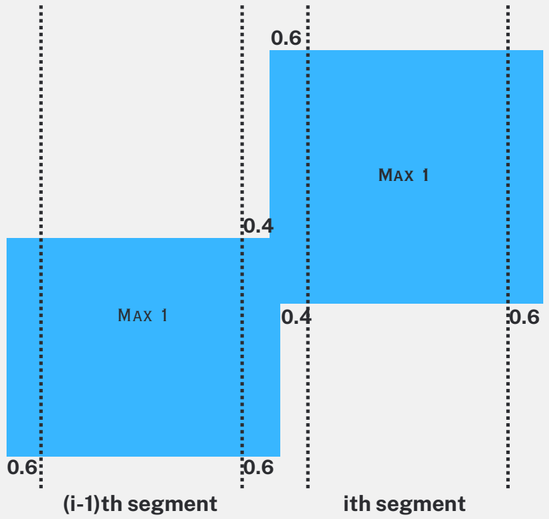
As shown below, in case of mismatch and selection of both subarrays, 8 extra vertices will be required.



How to form the polygon?

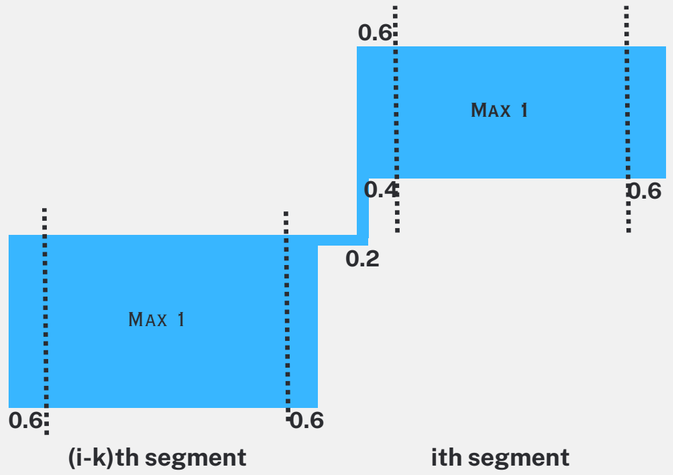
Magic of Floating Points

- Since floating points are allowed, we optimized the number of vertices by extending box edges, allowing them to naturally connect in some cases.
- We use a specific extension value of **0.6** to prevent unnecessary points from neighboring lines.
- This extension is applied on all four sides of the box, reducing complexity while maintaining accuracy.



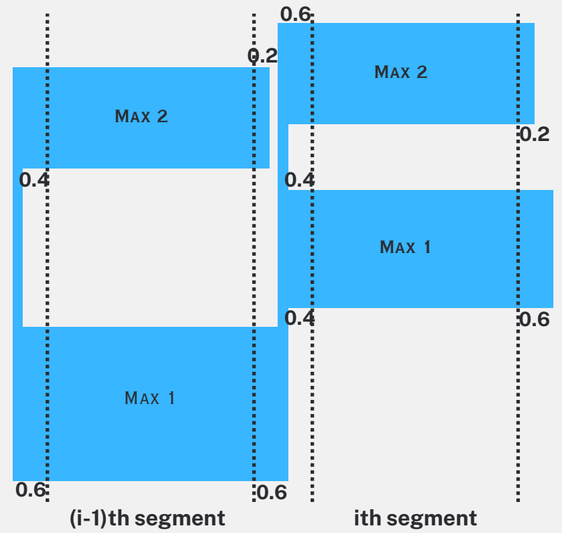
The Art of Pipelining

- In some cases, boxes may not connect even after a 0.6 extension.
- To address this, we create a pipeline from **Box 1 to Box 2** with a fixed width of **0.2** and only running between **(0.4 - 0.6)** points, ensuring no integer point is included.
- These pipelines are strategically designed to start and end at the outermost side of each box, eliminating extra vertices at their origin or endpoint.

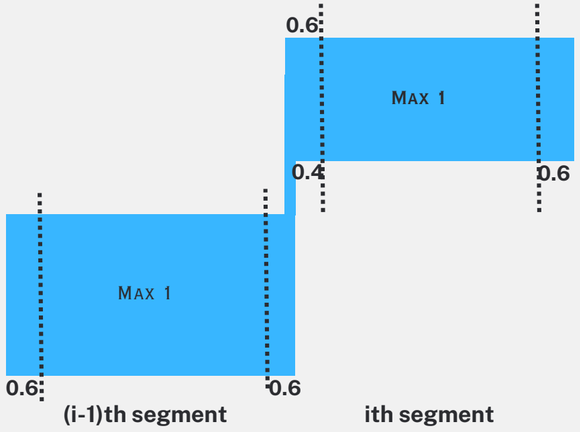


Mind the Gap!!

- Certain cases may still create cavities within the polygon. Example: Two subarrays in the **ith** column getting sandwiched between subarrays from **(i-1)th** and **(i+1)th** columns.
- To prevent this anomaly, we modify the second maximum subarray box by reducing one side's extension to **0.2 instead of 0.6**.
- This adjustment also ensures that no pipeline intersects the modified box, maintaining structural integrity.

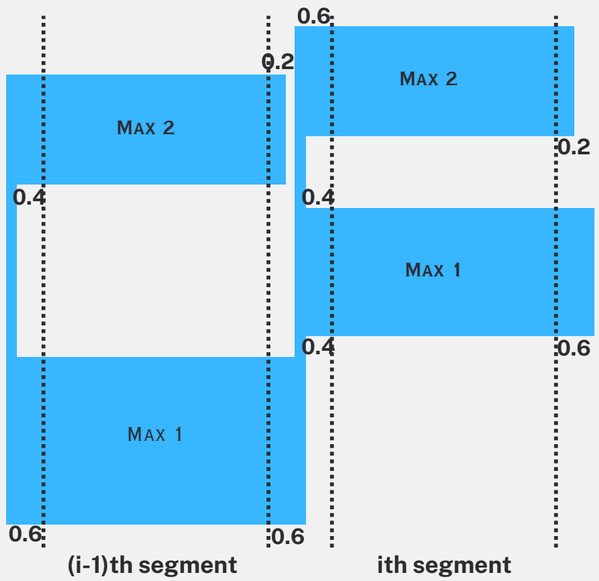


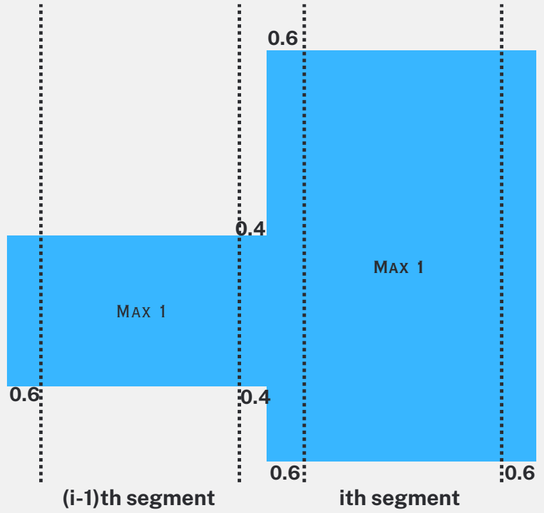
Some Cases for Adjacent Columns...



When only first maximums are considered in both

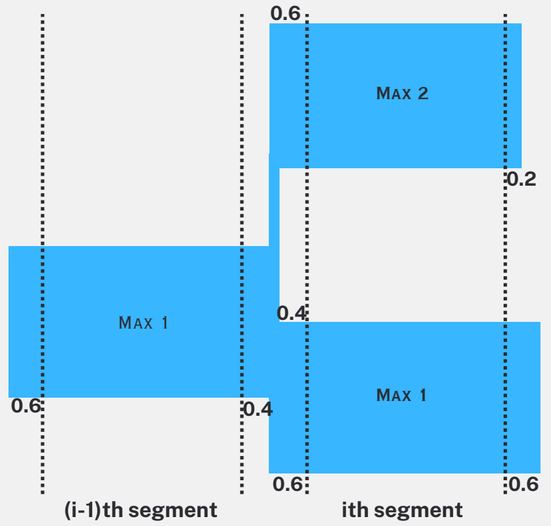
When first and second maximums are considered in both



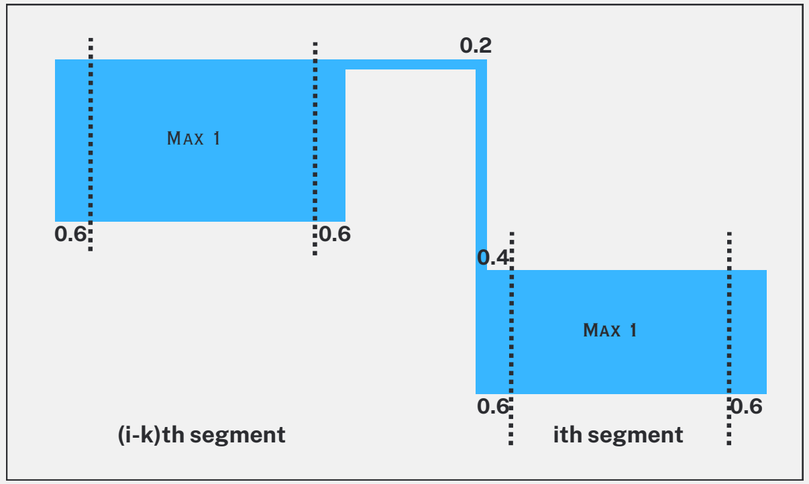


When both merge and one is smaller from both up and down

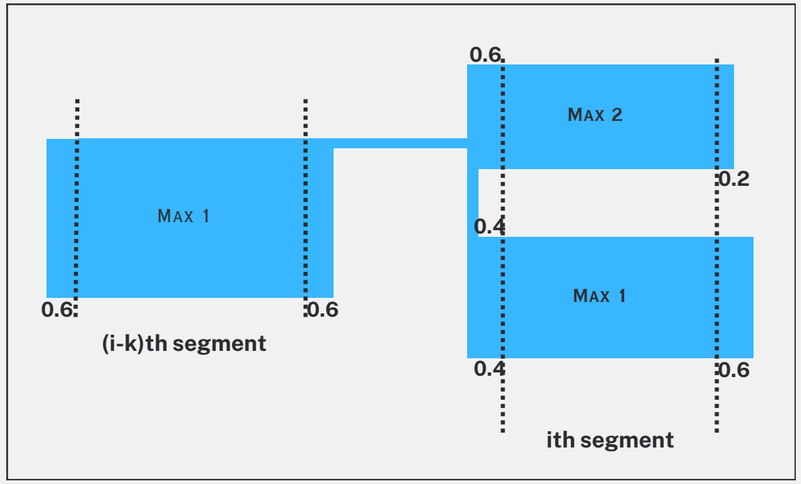
When two maximums in one and one in another is considered



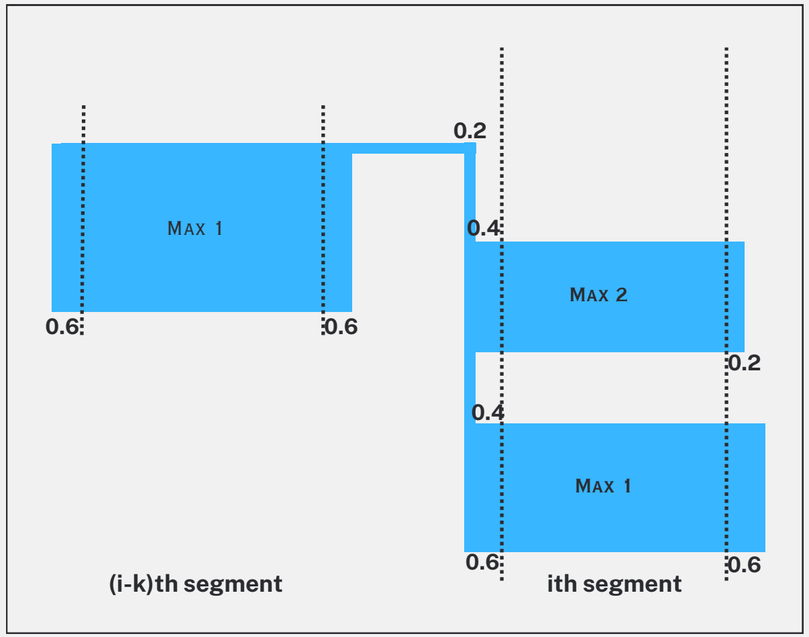
Some Cases for Non-Adjacent Columns...



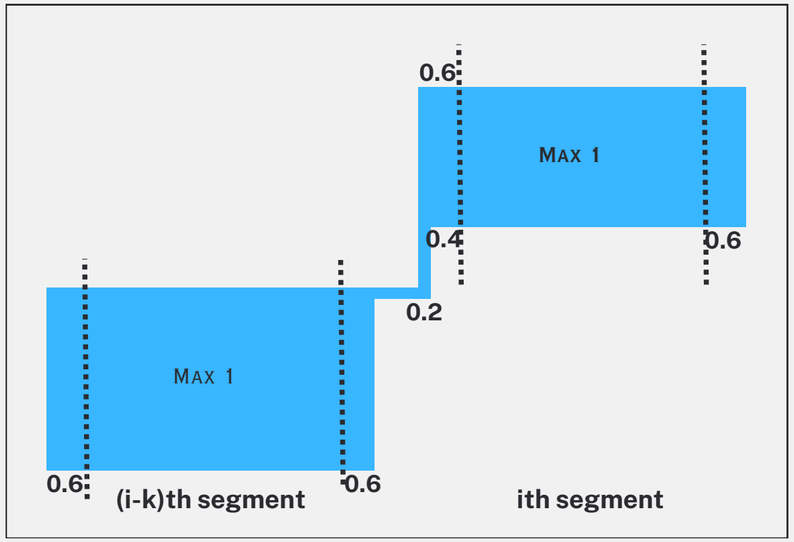
When current box is lower than the pipeline



When pipeline is in the middle of current two boxes



When pipeline is higher than the current two boxes

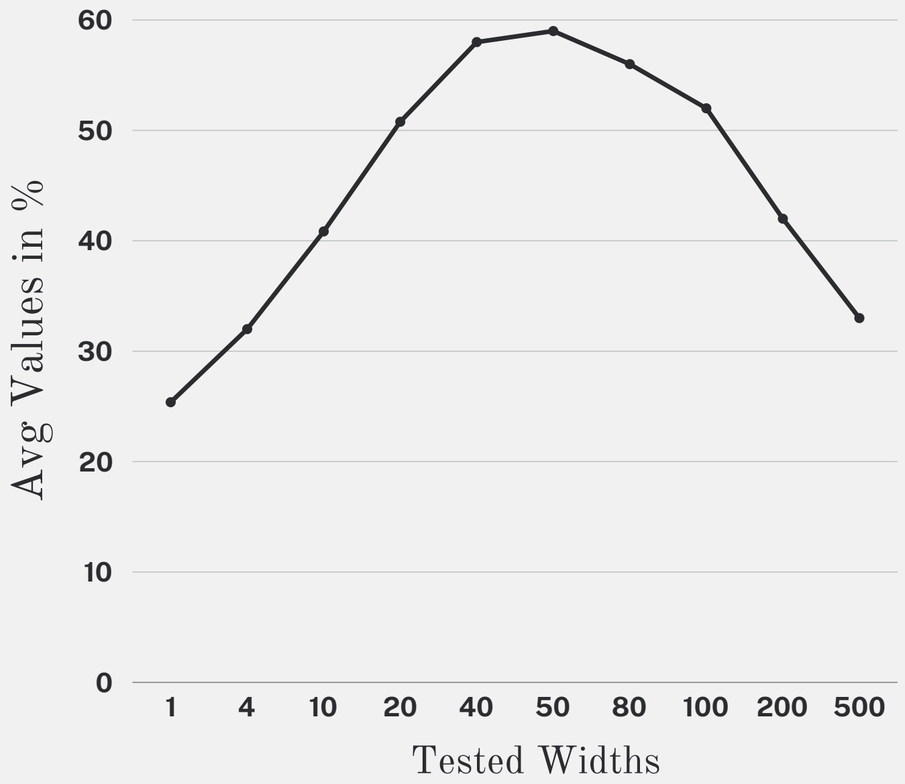


When current box is higher than the pipeline

Optimal Width Selection...

Testcase	Width = 1	Width = 10	Width = 20	Width = 40	Width = 50	Width = 80	Width = 100
input00	48.92	56.11	58.76	63.07	64.36	59.30	52.89
input01	11.74	29.94	41.7	53.96	55.27	51.87	50.12
input02	12.51	29.31	38.58	49.97	50.64	47.96	44.86
input03	29.51	38.07	40.47	43.46	45.79	43.71	37.83
input04	81.47	86.05	88.77	90.90	89.39	80.26	73.89
input05	12.10	28.12	36.44	45.80	46.24	44.96	41.34
input06	20.30	28.35	30.55	34.07	34.85	34.00	28.89
input07	10.64	51.41	90.77	97.15	97.15	96.96	96.94
input08	15.66	26.84	29.79	32.53	33.27	32.15	28.37
input09	11.11	34.44	52.18	71.79	71.38	69.11	68.39
Average:	25.396	40.864	50.801	58.27	58.834	56.028	52.352

- We tested the algorithm on multiples of 10,000 to identify 2-3 optimal widths.
- Due to time constraints, we selected the top two widths **(40 and 50)** from the graph, as they consistently provided the best results.

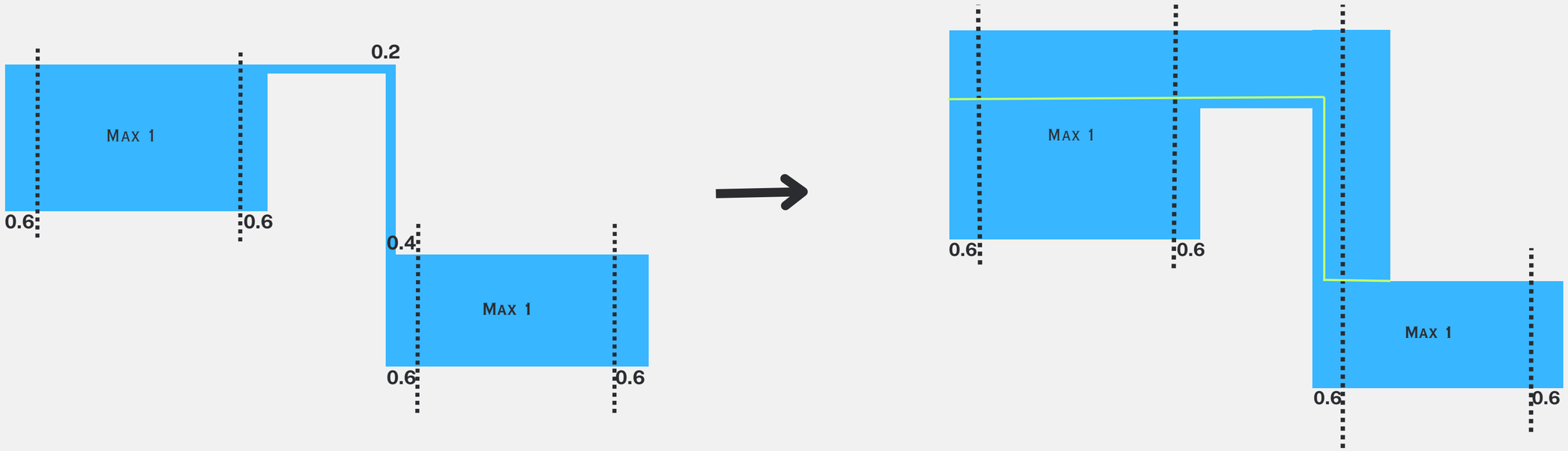


- Additionally, we included **width 1** as a third option to handle potential counter-cases and ensure robustness across different scenarios.
- Instead of columns, we've also taken in account rows by taking transpose of the grid giving better results.

Further Optimisations...

1. EXTENDING EDGES :

We can extend the current edges to maximize their contribution to the net positive value while ensuring that the total number of vertices in the polygon remains unchanged. This helps in improving the overall sum without altering the structural constraints.



2. USE REMAINING VERTICES/EDGES :

If the current number of vertices is less than 1000, we can connect additional positive regions to the polygon, further increasing the net positive value while staying within the vertex limit.

Results...

Testcase	Cost	Edges Used
input00	6634885	1000
input01	56006394	1000
input02	48638505	1000
input03	8629859	1000
input04	4525771	998
input05	42173102	1000
input06	12790877	1000
input07	152940938	646
input08	19176309	1000
input09	87006541	1000

Testcase	Cost	Edges Used
input10	254170	302
input11	5144694	1000
input12	2489061	1000
input13	31289420	1000
input14	3311463	1000
input15	8347390	1000
input16	2630183	998
input17	45654735	560
input18	3232890	1000
input19	7862330	1000

Final Total Output = **63.59%** of total positive values over all test cases

THANK YOU!

Team 57