COL 215 Software Assignment -1

Himanshi Bhandari(2023CS10888)

Mannat(2023CS10138)

22 August 2024

Problem statement

The primary goal is to arrange the gates in a way that minimizes the unused areas within the rectangular boundary. This involves finding an optimal or near-optimal placement for each gate. Gates must be placed without overlapping each other and also rotation of gates is not allowed.

Design Decisions

The following are the approaches we thought of and then finally the final approach

Approach 1-

- **Sorting gates in decreasing heights:** Initially we sorted the gates on the basis of the height(could also be done on basis of width).
- **Placing gates:** After comparing the sum of height of current gate and its previous gate with the maximum height gate(i.e first gate after sorting), we obtained whether to keep the rectangle beside the present rectangle or below it. If the obtained sum was less than largest height, we placed it above the present one, else it was placed next to the present one.(Similarly, we could use widths instead of heights)
- **Output:** Finally, the algorithm outputs the coordinates of each rectangle along with the dimensions of the bounding box. The coordinates indicate the position of each rectangle's top-left corner within the bounding box.
- This approach was very less efficient.

Approach 2-

- **Total Area and Row Width Calculation:** Firstly, we calculate the total area of the given gates, and then assigning the ceiling of the square root of the total area to the maximum width. This approach ensures that the bounding box is more or less square-shaped, which is generally optimal for packing.
- **Sorting Rectangles:** Then we sort the gates in decreasing order of their heights. Sorting by height first ensures that taller gates are placed side by side to gates of similar height, so the blank space reduces because the height of all gates in the row are as close as possible to the maximum height of the row.
- **Placing Rectangles:** Then we iterate over the sorted list of gates, attempting to place each rectangle in the current row if there's enough horizontal space left i.e. if the width of the gate+ width(space) filled in the row already<= maximum width. If the

gate can fit in the current row, its coordinates are set accordingly, and it is added to the row. If it doesn't fit in the current row, a new row is started below the previous one, and the rectangle is placed at the beginning of this new row. The x and y coordinates of each rectangle are updated based on their position in the row.

- **Bounding Box Dimensions:** Throughout the placement process, the algorithm keeps track of the maximum width used in any row and the total height accumulated by the rows. Once all rectangles are placed, these values represent the dimensions of the bounding box.

<u>Approach3(FINAL APPROACH)-</u>

We realized that the intuition of taking maximum width (width of bounding box) as the ceil of square root of total area may not give the minimum area always since the packing efficiency for smaller number of gates was not very high, so we iterated amongst all the possible widths of bounding box to find the minimum area.

- **Sorting Rectangles:** Then we sort the gates in decreasing order of their heights. Sorting by height first ensures that taller gates are placed side by side to gates of similar height, so the blank space reduces because the height of all gates in the row are as close as possible to the maximum height of the row. Thus, it minimizes gaps within each row, effectively reducing wasted space and making the overall packing more compact.

- **Row Width Calculation:** Firstly, we find the range of the width of bounding box. the maximum width of a gate amongst all gates and also the sum of the widths of all gates. This is done by calculating the maximum width among all the gates and the total sum of the widths of all gates. The algorithm iterates from the maximum width of a single gate to the total width of all gates. The rationale behind this is that the minimum possible bounding box width is constrained by the widest gate, while the maximum possible width occurs when all gates are placed in a single row. This iteration allows the algorithm to explore different potential bounding box widths, optimizing the arrangement of gates for each possible width.
    - **Placing Rectangles:** Then we iterate over the sorted list of gates, attempting to place each rectangle in the current row if there's enough horizontal space left i.e. if the width of the gate+ width(space) filled in the row already<= maximum width. If the gate can fit in the current row, its coordinates are set accordingly, and it is added to the row. If it doesn't fit in the current row, a new row is started below the previous one, and the rectangle is placed at the beginning of this new row. The x and y coordinates of each rectangle are updated based on their position in the row.

- **Bounding Box Dimensions:** Throughout the placement process, the algorithm keeps track of the maximum width used in any row (max_width_used) and the total height accumulated by the rows (curr_height). Once all rectangles are placed, these values represent the dimensions of the bounding box.
- **Calculation of minimum area:** We check if the area of bounding box is smaller than all the previous areas calculated whose minimum we have stored in min_area variable, if it is true then we update the value of min_area, width of bounding box , height of bounding box and store the coordinates of bottom left corner in best_coordinates.

- **Output:** Finally, the algorithm outputs the coordinates of each rectangle along with the dimensions of the bounding box with minimum area. The coordinates indicate the position of each rectangle's bottom-left corner within the bounding box, ensuring that they are packed in a space-efficient manner.

Time Complexity

Approach1-

- **Sorting Rectangles by Height**:
  While sorting the rectangles by their heights in decreasing order, the sorting algorithm typically used in Python is Timsort(which is used by Python's sort()), which has a time complexity of O(nlogn) in the worst case.

- **Placing Rectangles:** After this we check for each rectangle and compare its sum of heights with maximum height and places it accordingly. Since it iterates over each rectangle once its time complexity is O(n).

The **overall time complexity** of the algorithm is dominated by the sorting step, making it **O(nlogn)**.

Approach 2-

- **Total Area Calculation**:
  The algorithm calculates the total area of all gates by iterating through all the gates to sum up their areas. If there are n gates, this step takes O(n) time.

- **Sorting Rectangles by Height**:
  While sorting the rectangles by their heights in decreasing order, the sorting algorithm typically used in Python is Timsort(which is used by Python's sort()), which has a time complexity of O(nlogn) in the worst case.

- **Placing Rectangles into Rows:**

  After sorting, the algorithm iterates over each rectangle to place it into rows. For each rectangle, it checks if it can fit in the current row by comparing the cumulative width of the row plus the width of the current rectangle against the max_row_width(width of bounding box taken). If it fits, it is placed in the current row; otherwise, a new row is started. Therefore the time complexity is O(n) because each rectangle is considered once for placement.

- **Calculating Bounding Box Dimensions:**

  The bounding box dimensions are determined by keeping track of the maximum width used and the total height accumulated as rows are added. So the time complexity is O(1) because it simply returns the maximum values tracked during the rectangle placement.

The **overall time complexity** of the algorithm is dominated by the sorting step, making it **O(nlogn)**.

<u>**Final Approach-**</u>

- **Sorting Rectangles by Height**:
  While sorting the rectangles by their heights in decreasing order, the sorting algorithm typically used in Python is Timsort(which is used by Python's sort()), which has a time complexity of O(nlogn) in the worst case.

- **Packing Process**:
  - **Outer Loop**: Runs for each possible row width. In the worst case, this could be up to $O(n^2)$ iterations (from max_width to sum_of_widths), because max_width=100 and maximum value of sum_w can be 100(maximum value of width) * 1000(maximum value of gates)=1,00,000.
  - **Inner Loop**: Each gate is processed once per row width, so O(n) operations for each row width.
    The algorithm iterates over each rectangle to place it into rows. For each rectangle, it checks if it can fit in the current row by comparing the cumulative width of the row plus the width of the current rectangle against the max_row_width. If it fits, it is placed in the current row; otherwise, a new row is started. Therefore the time complexity is O(n) because each rectangle is considered once for placement.

  Overall, the packing process has a time complexity of $O(n \cdot k)$, where k is the number of potential row widths.

  - In the worst case, if max_width and sum_of_widths differ significantly, k can be close to $n^{1.67}$ rounding off to n^2. Thus, the time complexity of the packing process can approach $O(n^3)$.

Time complexity of input- Parsing the input file and initializing gates is O(n) because each gate is processed once to read its properties and create an instance.

Time complexity of output- Since writing the bounding box dimensions is constant time and writing the coordinates involves iterating through all gates, the overall complexity for the output phase is O(n)

Since terms nlogn and n can be ignored in front of n^3

Total Time complexity=O(n^3)

<u>TestCases:(test cases and outputs attatched with the pdf)</u>

<u>Test Case 1:</u> Minimum Input Values

This test case checks if the program can handle the smallest input values.

Here we got the packing efficiency as expected.

Input:

g1 100 100

Output:

bounding_box 100 100

g1 0 0

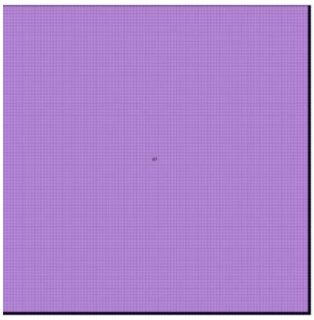

Figure 1 Test case1 output

<u>Test Case 2:</u>Same and maximum input values

This test case gives an input of 8 squares 0f side 100.

Here ,we got the packing efficiency as 100 as expected.

input	output
g1 100 100 g2 100 100 g3 100 100 g4 100 100 g5 100 100 g6 100 100 g7 100 100 g8 100 100	bounding_box 100 800 g1 0 0 g2 0 100 g3 0 200 g4 0 300 g5 0 400 g6 0 500 g7 0 600 g8 0 700

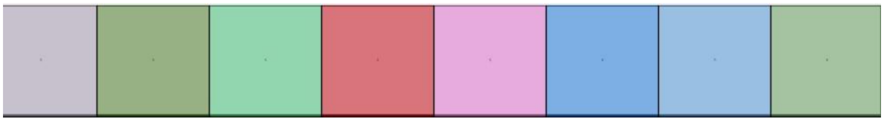

Figure 2Test case 2 output

<u>Test Cases 3:</u> 10 inputs with random values

This case gives input for 10 gates with different input values.

Here ,we got the packing efficiency as 82.

input	output
g1 66 80	bounding_box 141 297
g2 84 90	g10 0 0
g3 68 72	g2 32 0
g4 45 28	g1 0 95
g5 60 65	g3 66 95
g6 15 45	g5 0 175
g7 81 57	g7 60 175
g8 87 12	g6 0 240
g9 55 36	g9 15 240
g10 32 95	g4 70 240
	g8 0 285

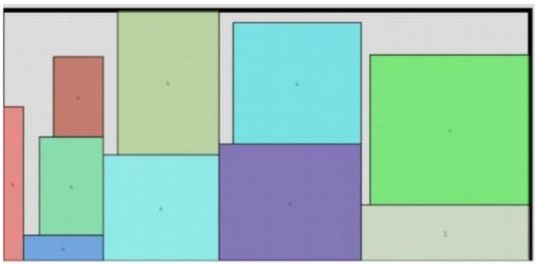

Figure 3 Test case 3 output

<u>Test Cases4:</u> 50 inputs with random values

This case gives input for 50 gates with different input values.

Here ,we got the packing efficiency as 87 .

input			output		
g1 89 30	g17 22 75	g33 86 15	bounding_box 187 702	g39 0 337	g24 76 585
g2 47 69	g18 45 58	g34 58 2	g45 0 0	g22 83 337	g23 97 585
g3 48 6	g19 28 3	g35 76 29	g31 31 0	g25 93 337	g29 0 614
g4 49 67	g20 18 57	g36 80 78	g6 90 0	g18 0 403	g32 55 614
g5 17 11	g21 62 13	g37 36 17	g12 0 95	g20 45 403	g44 0 640
g6 88 88	g22 10 63	g38 67 22	g11 16 95	g9 63 403	g14 59 640
g7 47 73	g23 59 27	g39 83 66	g43 43 95	g42 0 461	g38 104 640
g8 67 49	g24 21 28	g40 63 34	g27 92 95	g8 65 461	g28 0 665

g9 93 53	g25 69 60	g41 81 77	g30 0 183	g47 132 461	g48 31 665
g10 22 72	g26 51 35	g42 65 50	g36 25 183	g50 149 461	g37 41 665
g11 27 87	g27 93 79	g43 49 85	g41 105 183	g15 0 511	g33 77 665
g12 16 88	g28 31 20	g44 59 25	g17 0 262	g49 67 511	g13 163 665
g13 12 14	g29 55 26	g45 31 95	g7 22 262	g26 121 511	g21 0 685
g14 45 22	g30 25 79	g46 78 4	g10 69 262	g16 0 551	g5 62 685
g15 67 40	g31 59 92	g47 17 45	g2 91 262	g40 29 551	g3 79 685
g16 29 34	g32 96 25	g48 10 19	g4 138 262	g1 92 551	g46 0 698
		g49 54 39		g35 0 585	g19 78 698
		g50 1 45			g34 106 698

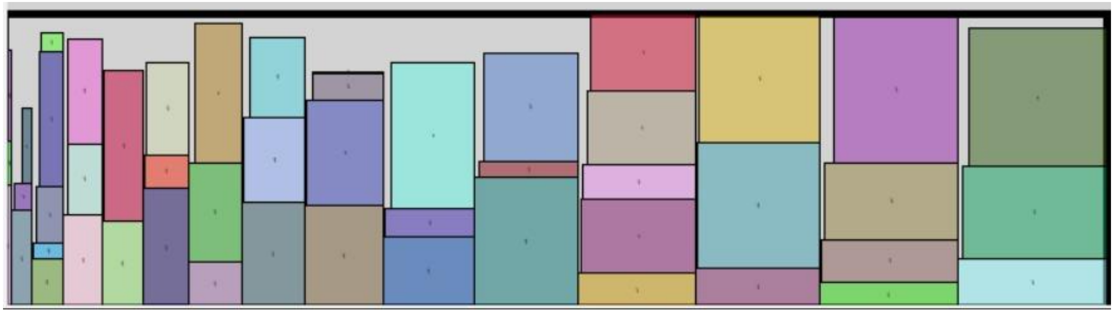

Figure 4 Test case 4 output

<u>Test Case 5:</u> 100 inputs with random values

This case gives input for 100 gates with different input values.

Here ,we got the packing efficiency as 89 .

input			output		
g1 45 1	g34 6 98	g67 48 2	bounding_box 306 727	g64 202 276	g65 118 589
g2 92 23	g35 57 28	g68 21 19	g22 0 0	g85 268 276	g61 194 589
g3 55 91	g36 60 45	g69 52 66	g12 23 0	g46 300 276	g79 238 589
g4 35 88	g37 43 62	g70 21 15	g34 30 0	g90 0 354	g96 285 589
g5 21 85	g38 1 21	g71 44 11	g57 36 0	g52 28 354	g30 0 632
g6 44 60	g39 36 7	g72 20 47	g44 85 0	g69 102 354	g97 22 632
g7 12 9	g40 22 57	g73 81 64	g84 88 0	g75 154 354	g98 51 632
g8 12 50	g41 34 11	g74 96 79	g25 99 0	g87 158 354	g18 139 632
g9 27 78	g42 16 2	g75 4 66	g56 184 0	g73 185 354	g59 206 632
g10 36 44	g43 47 75	g76 16 56	g92 198 0	g78 266 354	g77 231 632
g11 3 55	g44 3 97	g77 18 29	g47 204 0	g17 0 422	g35 249 632
g12 7 99	g45 19 42	g78 25 64	g53 256 0	g99 70 422	g86 0 664
g13 77 57	g46 4 69	g79 47 37	g62 0 100	g37 155 422	g49 52 664
g14 70 7	g47 52 92	g80 76 15	g3 55 100	g83 198 422	g60 146 664
g15 66 90	g48 38 1	g81 45 78	g15 110 100	g33 221 422	g2 0 692
g16 17 81	g49 94 27	g82 45 7	g4 176 100	g6 262 422	g24 92 692
g17 70 63	g50 66 78	g83 23 62	g32 211 100	g13 0 485	g100 121 692
g18 67 29	g51 75 44	g84 11 95	g5 249 100	g40 77 485	g23 133 692
g19 44 77	g52 74 66	g85 32 70	g20 0 192	g76 99 485	g38 134 692
g20 39 84	g53 18 92	g86 52 28	g29 39 192	g88 115 485	g91 135 692
g21 4 49	g54 10 84	g87 27 66	g54 55 192	g11 126 485	g68 162 692
g22 23 100	g55 39 47	g88 11 56	g26 65 192	g63 129 485	g70 183 692
g23 1 22	g56 14 93	g89 68 1	g94 80 192	g27 215 485	g80 204 692

g24 29 23 g25 85 94 g26 15 83 g27 74 51 g28 54 1 g29 16 84 g30 22 32 g31 13 44 g32 38 86 g33 41 61	g57 49 98 g58 26 40 g59 25 29 g60 71 26 g61 44 38 g62 55 92 g63 86 52 g64 66 74 g65 76 39 g66 73 43	g90 28 68 g91 27 20 g92 6 93 g93 22 80 g94 63 83 g95 47 1 g96 21 34 g97 29 32 g98 88 30 g99 85 63 g100 12 23	g16 143 192 g93 160 192 g74 182 192 g9 278 192 g50 0 276 g81 66 276 g19 111 276 g43 155 276	g8 289 485 g21 301 485 g55 0 542 g72 39 542 g36 59 542 g10 119 542 g31 155 542 g51 168 542 g66 0 589 g45 73 589 g58 92 589	g41 0 715 g71 34 715 g7 78 715 g14 90 715 g39 160 715 g82 196 715 g42 241 715 g67 257 715 g1 0 726 g28 45 726 g48 99 726 g89 137 726 g95 205 726

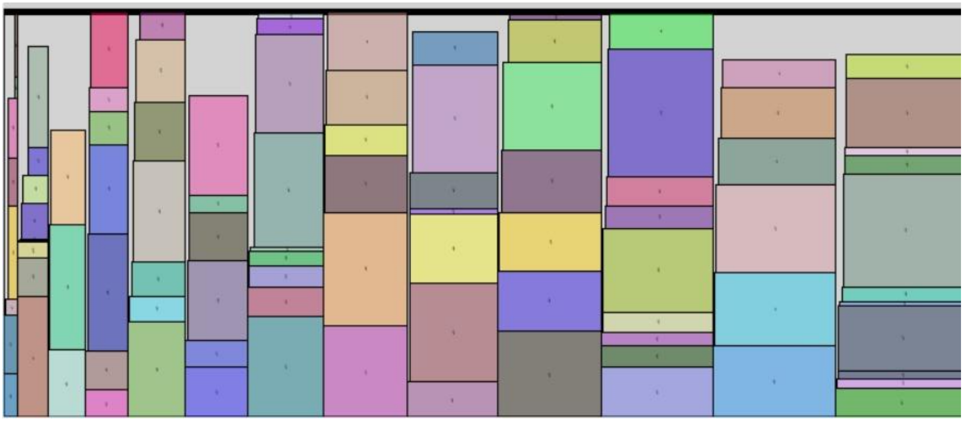

Figure 5 Test case 5 output

<u>Test Case 6:</u> 500 inputs with random values

This case gives input for 500 gates with different input values.

Here ,we got the packing efficiency as 94 .

<u>Test Case 7:</u> 1000 inputs with random values

This case gives input for 1000 gates with different input values.

Here ,we got the packing efficiency as 96.

<u>Public test cases:</u>

Test case 1:

input	output
g1 3 10 g2 8 3 g3 6 6	bounding_box 9 13 g1 0 0 g3 3 0 g2 0 10

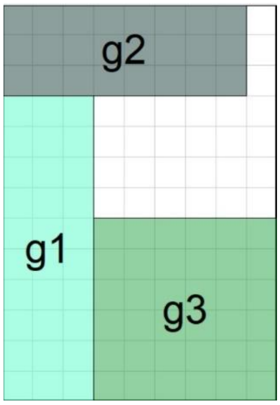

Figure 6 public test case 1 output

Test case 2:

input	output
g1 3 4 g2 5 2 g3 2 3	bounding_box 5 6 g1 0 0 g3 3 0 g2 0 4

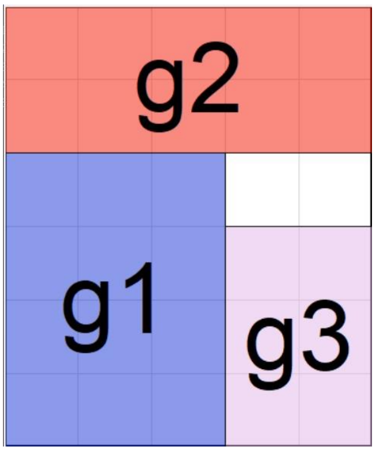

Figure 7 public test case 2 output

Test cases 3:

input	output
g1 10 10 g2 20 5 g3 5 20 g4 15 10 g5 10 15 g6 25 5 g7 5 25 g8 30 10 g9 10 30	bounding_box 45 50 g9 0 0 g7 10 0 g3 15 0 g5 20 0 g1 30 0 g4 0 30 g8 15 30 g2 0 40

g10 35 5	g6 20 40 g10 0 45

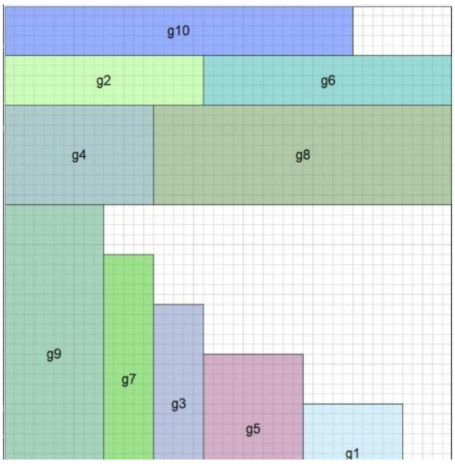

Figure 8 public test case 3 output

Test case 4:

input	output
g1 4 5 g2 6 2 g3 2 6 g4 3 4 g5 5 3	bounding_box 6 15 g3 0 0 g1 2 0 g4 0 6 g5 0 10 g2 0 13

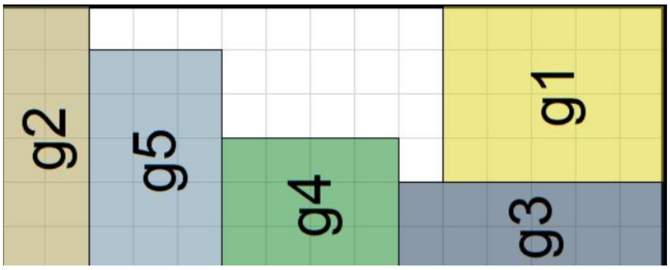

Figure 9 public test case 4 output

Test case 5

input	output
g1 3 4 g2 5 2 g3 2 2 g4 3 2 g5 12 2 g6 10 2 g7 3 7 g8 3 3 g9 5 3 g10 6 2	bounding_box 15 42 g13 0 0 g22 5 0 g20 7 0 g35 13 0 g14 0 13 g26 3 13 g7 4 13 g15 7 13 g21 10 13

g11 3 2	g17 11 13
g12 3 2	g1 0 22
g13 5 13	g34 3 22
g14 3 9	g8 4 22
g15 3 6	g9 7 22
g16 3 3	g16 12 22
g17 3 5	g19 0 26
g18 8 2	g24 8 26
g19 8 3	g25 11 26
g20 6 10	g29 14 26
g21 1 6	g2 0 29
g22 2 11	g3 5 29
g23 3 2	g4 7 29
g24 3 3	g5 0 31
g25 3 3	g6 0 33
g26 1 8	g10 0 35
g27 5 1	g11 6 35
g28 1 1	g12 9 35
g29 1 3	g18 0 37
g30 5 2	g23 8 37
g31 5 1	g30 0 39
g32 6 2	g32 5 39
g33 1 1	g27 0 41
g34 1 4	g28 5 41
g35 2 10	g31 6 41
	g33 11 41

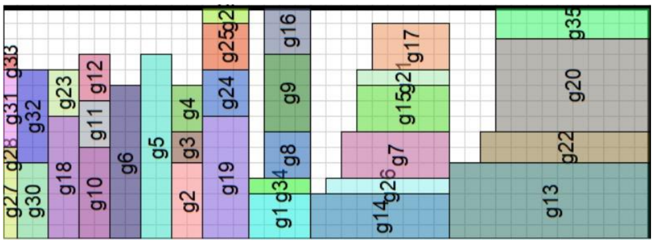

Figure 10 public test case 5 output

COMPARISON BETWEEN APPROACHES-

The time complexity of first approach was O(nlogn) but its packing efficiency was very less. The second approach also has time complexity O(nlogn) but its packing efficiency better than the first approach. The packing efficiency varied from 60-100% (efficiency >90%for test cases involving 500-1000 gates) for the second approach whereas our final approach has packing efficiency 75-100% (efficiency >94%for test cases involving 500-1000 gates), but the time complexity is better for second approach. We prioritized packing efficiency as the difference between times was not much.

CONCLUSION-

We conclude that our code demonstrates strong performance for a large number of gates, making it highly suitable for practical applications. The packing accuracy is consistently high, with nearly 95% or greater accuracy observed for test cases involving 500-1000 gates. For most test cases, the accuracy ranges between 80% and 100%, with only a few cases dipping slightly above 75%.

We observed that some space remains in the rows where gates of smaller widths could be placed side by side, and smaller height gates could be positioned above the placed gates in the row. However, for a large number of gates, these blank spaces are minimal, further reinforcing the efficiency of the packing.

The time complexity of the algorithm is O(n3), indicating that the algorithm efficiently produces outputs in a reasonable time frame, even as the number of gates increases.