EE6332 - ENDSEM PROJECT — Gate Sizing using Geometric Programming and Branch and Bound

Himanshu Rajnish Borkar - EE22B070

22/05/2025

1 Introduction

Gate sizing is a critical optimization problem in digital circuit design, aiming to meet performance specifications such as delay while minimizing area or power. Due to the combinatorial nature of selecting discrete gate sizes, the problem is generally hard. This report presents a two-phase approach:

- **Phase 1:** Solve the relaxed problem using Geometric Programming (GP) to obtain a continuous solution.
- **Phase 2:** Use a Branch and Bound (BnB) algorithm to convert the continuous solution into a feasible discrete one.

2 Phase 1: Continuous Relaxation with Geometric Programming

In the first phase, we model the gate sizing problem as a geometric program:

- **Variables:** Gate sizes x_i and arrival times A_i
- **Objective:** Minimize total gate area $\sum x_i$
- **Constraints:** Timing constraints between gates and an upper bound on total delay:

$$T_{\text{wall}} \leq T_{\text{spec}} = 1.3 \cdot T_{\text{wall\_solved}}$$

The solution yields optimal (but fractional) gate sizes and timing information. These values serve as a lower bound and a guide for the next phase.

Plots and Visualizations

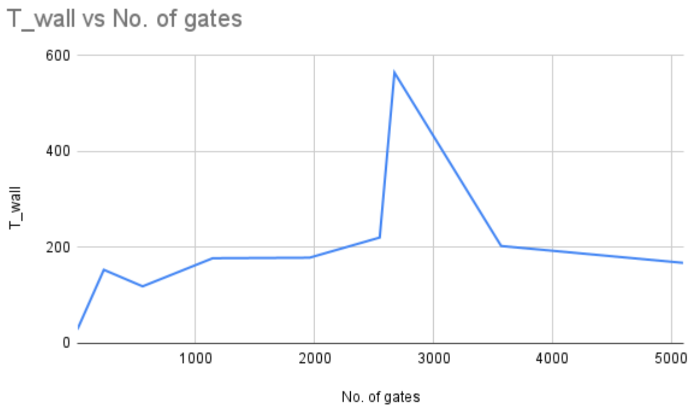
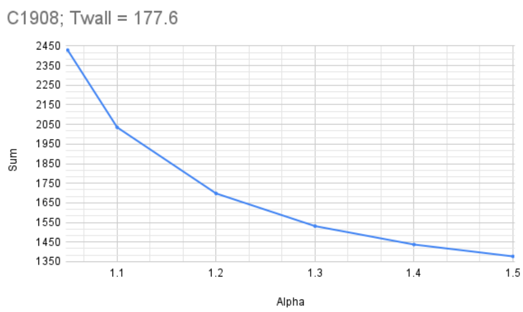
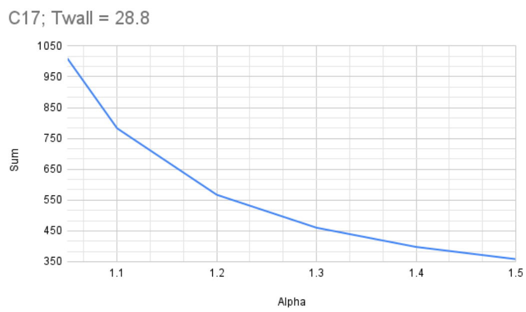


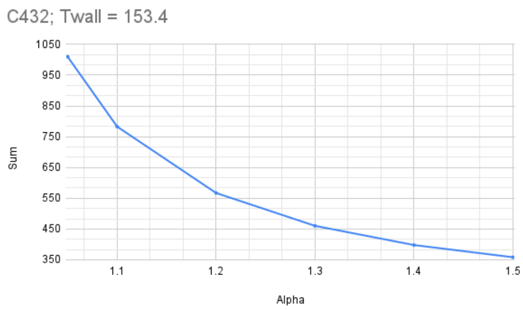
Figure 1: T_{wall} vs Number of Gates



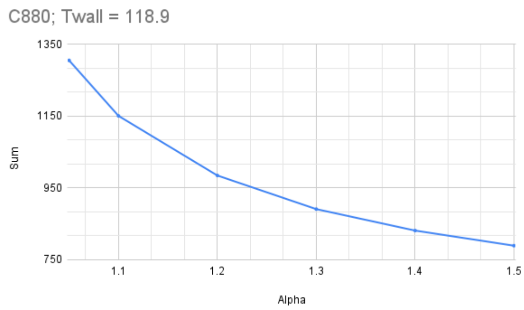
(a) A_{min} vs T_{spec} (c1908)



(b) A_{min} vs T_{spec} (c17)

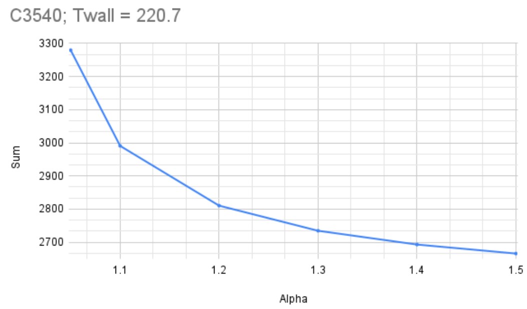


(c) A_{min} vs T_{spec} (c432)

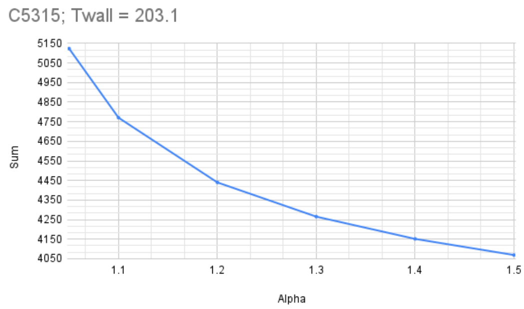


(d) A_{min} vs T_{spec} (c880)

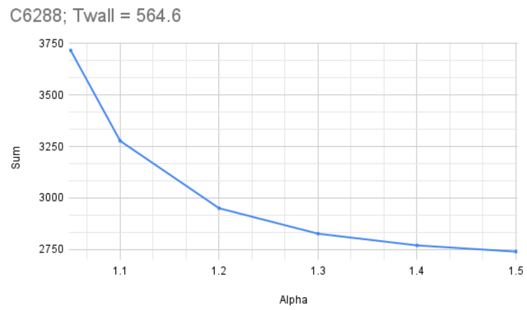
Figure 2: A_{min} vs T_{spec}



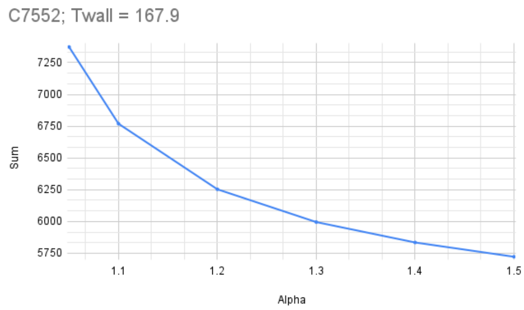
(a) A_{min} vs T_{spec} (c3540)



(b) A_{min} vs T_{spec} (c5315)



(c) A_{min} vs T_{spec} (c6288)



(d) A_{min} vs T_{spec} (c7552)

Figure 3: A_{min} vs T_{spec}

Column 1	# Twall	# 1.05	# 1.1	# 1.2	# 1.3	# 1.4	# 1.5
c432	153.4	1009.9	782.8	567.1	460.3	397.9	358.3
c17	28.8	199.1	182.7	156.3	136.1	120.2	107.4
c880	118.9	1305.5	1150.8	984.8	891.2	831.4	789.3
c1908	177.6	2429.1	2035.2	1698	1531.5	1437.5	1377.1
c3540	220.7	3279.6	2990.7	2810.4	2734.6	2693	2665.8
c5315	203.1	5123.7	4770.5	4440	4265.3	4151.6	4069.3
c6288	564.6	3716.9	3277.1	2949.7	2825.9	2769.1	2738.7
c7552	167.9	7373	6769.1	6253.3	5995.4	5834.9	5722.6

Figure 4: Dataset

Analysis Questions

Q1. Look at the critical path and analyze the gate sizes on the critical path. What do you observe on noncritical paths?

Ans: The critical path contains the gates with larger sizes to reduce the delay as much as possible, whereas the non-critical paths have smaller gate sizes in order to reduce the total area and save power.

Q2. Analyze the effect of discretizing the continuous solution. Can you think of how you can identify a discrete solution is likely to satisfy the timing constraint?

Ans: One way to discretize the solution is to choose a more relaxed T_{spec}, such as $1.35 \times T_{\text{wall}}$ instead of $1.3 \times T_{\text{wall}}$, and then round the continuous solution up to the ceiling value to obtain a discrete solution. In this way, even with the relaxed T_{spec}, there is no timing violation.

Q3. If you were a standard cell designer and you had to identify the drive strength of various gates (INV, NAND2-4, NOR2-4), what would you decide?
Ans: We consider each gate, for example the NAND2 gate and plot a histogram of the sizes used in the circuit. This distribution typically resembles a bell-curve, and the most frequent size is chosen as the standard discrete drive strength.

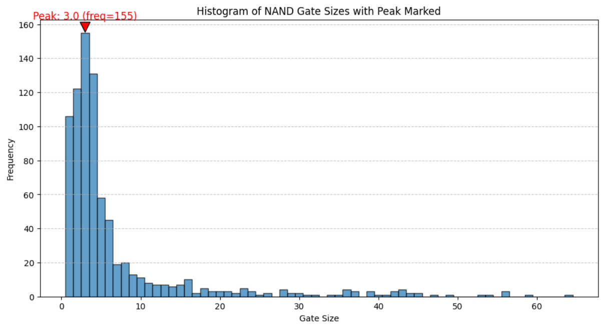


Figure 5: Histogram of gate sizes for NAND2 gates

3 Phase 2: Discrete Optimization with Branch and Bound

While the continuous solution from Phase 1 is optimal in theory, practical constraints require gate sizes to be discrete. To address this, we use a Branch and Bound (BnB) algorithm tailored to the problem structure.

3.1 Key Concepts

1. Initial Setup: Use the solution from Phase 1 to initialize the BnB search and set area bounds.

2. Branching Rule: Select the variable (gate size) with the largest fractional part and highest priority to branch on.

3. Bounding: If a solution exceeds the known best area or violates feasibility, prune that branch.

4. Termination: Stop when a valid integer solution is found within bounds or when a time/resource limit is reached.

3.2 Branch and Bound Python Code

```
def branch_and_bound(constraints_input):
    num_vars = N
    best_solution = [np.inf] * num_vars
    best_area = np.inf
    fewest_fractions = np.inf
    best_partial_solution = [np.inf] * num_vars
    best_partial_constraints = []

    timeout = min(100 * num_vars, 1800)
    start_time = time.time()

    def recurse(current_constraints, last_fraction_count=np.inf):
        nonlocal best_solution, best_area, fewest_fractions
        nonlocal best_partial_solution, best_partial_constraints

        if time.time() - start_time > timeout:
            return

        try:
            candidate, area = solve_model(current_constraints)
        except (PrimalInfeasible, UnknownInfeasible, Infeasible,
                subprocess.CalledProcessError, Exception):
            return

        if area > best_area or area > 1.4 * initial_area:
            return

        fraction_count = sum(
            abs(round(candidate[i]) - candidate[i]) > tolerance
               for i in range(num_vars)
        )
        print(fraction_count, area)

        if fraction_count < fewest_fractions:
            fewest_fractions = fraction_count
            best_partial_solution = candidate.copy()
            best_partial_constraints = current_constraints.copy()

        if is_integer_vector(candidate, num_vars):
            if area < best_area:
                best_solution = candidate
                best_area = area
            return

        target_index = None
        max_fraction = -1
        for i in range(num_vars):
            frac_part = abs(round(candidate[i]) - candidate[i])
            if frac_part > tolerance and round(candidate[i], 1):
                if critical_flag[i] and frac_part > max_fraction:
```

```
                    max_fraction = frac_part
                    target_index = i

        if target_index is None:
            for i in range(num_vars):
                frac_part = abs(round(candidate[i]) - candidate[i])
                if frac_part > tolerance and frac_part > max_fraction and round(candidate[i], 1):
                    max_fraction = frac_part
                    target_index = i

        if target_index is None:
            return

        low_val = int(np.floor(candidate[target_index]))
        high_val = int(np.ceil(candidate[target_index]))

        left_constraints = current_constraints.copy()
        left_constraints.append(x[target_index] <= low_val)
        recurse(left_constraints, fraction_count)

        right_constraints = current_constraints.copy()
        right_constraints.append(x[target_index] >= high_val)
        recurse(right_constraints, fraction_count)

    recurse(constraints_input)

    return best_solution, best_area, best_partial_solution, best_partial_constraints
```

Listing 1: Branch and Bound Implementation

3.3 Final Output

After the BnB algorithm terminates, the final output includes:

- Best integer solution found (gate sizes x_i)
- Total area of the solution
- Optionally, a rounded version of the best partial (fractional) solution

4 Conclusion

The two-phase gate sizing approach combines the efficiency of Geometric Programming with the rigor of Branch and Bound to provide discrete, high-quality solutions that meet timing constraints. This strategy effectively balances optimality with feasibility in practical digital design workflows.