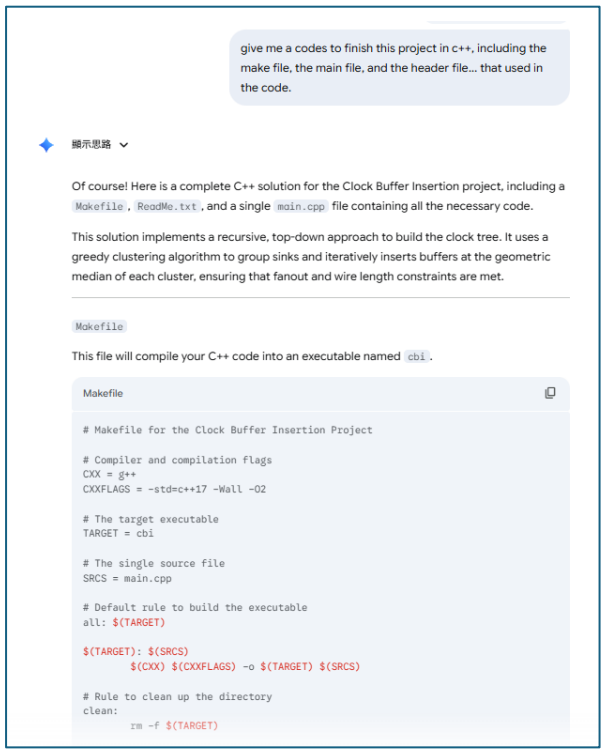


## 1. Introduction

The goal of this project was to design and implement a C++ program for clock buffer insertion, a key step in Clock Tree Synthesis (CTS). The primary objectives were to satisfy fanout and wire length constraints while minimizing total wire length and clock skew. Throughout this project, I used the AI assistant Gemini as a collaborative tool for brainstorming algorithms, generating C++ code, and debugging complex issues. This report details the evaluation of several methods and the justification for my final design choice.



## 2. Methodological Exploration and Evaluation

My final design choice was the result of a rigorous process of implementing and evaluating several different algorithms. All implemented methods produced valid results that worked for the test cases, but they revealed important trade-offs between wire length and clock skew.

- **Method 1: Method of Means and Medians (MMM):** My initial robust implementation was based on the classic MMM algorithm. This method

produces structurally balanced trees and served as a strong performance benchmark, delivering good, consistent results across all test cases.

but I want to split the functions to multiple file

顯示思路

Of course. Splitting the code into multiple files is great for organization and scalability. Here's the refactored project, now divided into a header file for declarations (`cbi.h`), a source file for implementations (`cbi.cpp`), and the main driver (`main.cpp`). The `Makefile` is also updated accordingly.

cbi.h (Header File)

This file declares the data structures and the `ClockTreeSynthesizer` class, serving as the public interface for your module.

```
C++  
  
#ifndef CBI_H  
#define CBI_H  
  
#include <iostream>  
#include <vector>  
#include <string>  
#include <map>  
#include <set>  
#include <limits>  
  
// Represents a 2D integer coordinate  
struct Point {  
    int x = 0;  
    int y = 0;  
    bool operator<(const Point& other) const;  
};  
  
// Enum to identify the type of node in the clock tree  
enum NodeType { SRC, SINK, BUFFER };  
  
// Represents a node in the clock tree (source, sink, or buffer)  
struct Node {  
    std::string id;  
    NodeType type;  
    Point pos;  
};  
  
// ClockTreeSynthesizer class declaration  
class ClockTreeSynthesizer {  
public:  
    // Constructor  
    ClockTreeSynthesizer();  
    // Destructor  
    ~ClockTreeSynthesizer();  
    // Parse input file  
    bool parseInput(const std::string& filename);  
    // Write output file  
    void writeOutput(const std::string& filename);  
    // Populate level map  
    void populateLevelMap(Node* node);  
    // Cleanup  
    void cleanup();  
    // Print metrics  
    void printMetrics() const;  
    // Manhattan distance  
    long long manhattanDistance(Point p1, Point p2) const;  
    // Calculate cluster wire length  
    long long calculateClusterWireLength(Point center, const std::vector<Node*>& cluster);  
};  
  
#endif
```

問問 Gemini

+ 工具

cbi.cpp (Implementation File)

This file provides the logic for all the functions declared in `cbi.h`.

```
C++  
  
#include "cbi.h"  
#include <fstream>  
#include <sstream>  
#include <cmath>  
#include <algorithm>  
  
// Operator implementation for the Point struct to enable its use in std::set  
bool Point::operator<(const Point& other) const {  
    if (x != other.x) return x < other.x;  
    return y < other.y;  
}  
  
// Destructor to clean up dynamically allocated nodes  
ClockTreeSynthesizer::~ClockTreeSynthesizer() {  
    cleanup();  
}  
  
bool ClockTreeSynthesizer::parseInput(const std::string& filename) {  
    std::ifstream inFile(filename);  
    if (!inFile.is_open()) {  
        std::cerr << "Error: Could not open input file " << filename << std::endl;  
        return false;  
    }  
  
    std::string line, token;  
    while (getline(inFile, line)) {  
        if (line.empty() || line[0] == '#') continue;  
        std::stringstream ss(line);  
        ss >> token;  
  
        if (token == ".limit") {  
            getline(inFile, line);  
            std::stringstream(line) >> token >> maxFanout;  
            getline(inFile, line);  
            std::stringstream(line) >> token >> maxLength;  
        } else if (token == ".pin") {  
            int numPins;  
            ss >> numPins;  
  
            int x, y;  
            inFile >> x >> y;  
            srcNode = new Node{"SRC", SRC, {x, y}};  
        }  
    }  
  
    return true;  
}  
  
void ClockTreeSynthesizer::writeOutput(const std::string& filename) {  
    std::ofstream outFile(filename);  
    if (!outFile.is_open()) {  
        std::cerr << "Error: Could not open output file " << filename << std::endl;  
        return;  
    }  
  
    outFile << ".buffer " << buffers.size() << " # number of clock buffers" << std::endl;  
    for (const auto& buffer : buffers) {  
        outFile << buffer->id << " " << buffer->pos.x << " " << buffer->pos.y << " "  
    }  
    outFile << ".e" << std::endl;  
  
    populateLevelMap(srcNode);  
    outFile << ".level " << levelMap.size() - 1 << " # number of tree levels" << std::endl;  
    for (size_t i = 0; i < levelMap.size() - 1; ++i) {  
        outFile << (i + 1) << " " << levelMap[i].size() << " "  
        for (const auto& parent_node : levelMap[i]) {  
            if (!parent_node->children.empty()) {  
                outFile << parent_node->id << " (" << "  
                for (size_t j = 0; j < parent_node->children.size(); ++j) {  
                    outFile << parent_node->children[j]->id << " (j == parent_node->children[j] - 1) " << "  
                }  
                outFile << ") " << "  
            }  
        }  
        outFile << "# hierarchy" << std::endl;  
    }  
    outFile << ".e" << std::endl;  
}  
  
void ClockTreeSynthesizer::printMetrics() const {  
    std::cout << "T_max: " << maxArrivalTime << "  
    << "T_min: " << minArrivalTime << "  
    << "W_cbi: " << totalWireLength << std::endl;  
}  
  
long long ClockTreeSynthesizer::manhattanDistance(Point p1, Point p2) const {  
    return std::abs(p1.x - p2.x) + std::abs(p1.y - p2.y);  
}  
  
long long ClockTreeSynthesizer::calculateClusterWireLength(Point center, const std::vector<Node*>& cluster) {  
    long long length = 0;  
    for (const auto& node : cluster) {  
        length += manhattanDistance(center, node->pos);  
    }  
    return length;  
}
```

Gemini

工具

main.cpp (Driver File)

This file provides the logic for all the functions declared in `cbi.h`.

```
C++  
  
#include "cbi.h"  
#include <iostream>  
#include <vector>  
#include <string>  
#include <map>  
#include <set>  
#include <limits>  
  
// Main driver function  
int main() {  
    ClockTreeSynthesizer synthesizer;  
    if (!synthesizer.parseInput("input.txt")) {  
        return 1;  
    }  
    synthesizer.writeOutput("output.txt");  
    synthesizer.printMetrics();  
    synthesizer.cleanup();  
    return 0;  
}
```

Gemini

工具

main.cpp (Driver File)

This file is now much cleaner, containing only the `main` function that drives the program.

```
C++
#include "cbi.h"
#include <iostream>

// Main function to drive the program
int main(int argc, char* argv[]) {
    // Check for correct command-line arguments
    if (argc != 3) {
        std::cerr << "SYNOPSIS for CBI" << std::endl;
        std::cerr << "%s cbi INPUT_FILE OUTPUT_FILE" << std::endl;
        return 1;
    }

    std::string inputFile = argv[1];
    std::string outputFile = argv[2];

    ClockTreeSynthesizer cts;

    // Execute the clock tree synthesis process
    if (cts.parseInput(inputFile)) {
        cts.buildTree();
        cts.writeOutput(outputFile);
        cts.printMetrics();
    } else {
        return 1; // Parsing failed
    }

    return 0;
}
```

C++

```
public:
    ClockTreeSynthesizer() = default;
    ~ClockTreeSynthesizer();

    bool parseInput(const std::string& filename);
    void buildTree();
    void writeOutput(const std::string& filename);
    void printMetrics() const;

private:
    // Input constraints and data
    int maxFanout;
    int maxLength;
    Node* srcNode = nullptr;
    std::vector<Node*> sinks;

    // Generated data
    std::vector<Node*> buffers;
    std::map<int, std::vector<Node*>> levelMap;
    std::set<Point> occupiedCoordinates;

    // Final calculated metrics
    long long totalWireLength = 0;
    long long maxArrivalTime = 0;
    long long minArrivalTime = std::numeric_limits<long long>::max();

    // Core recursive function to build the tree
    void recursiveBuild(Node* parent, std::vector<Node*> targets);

    // Helper functions
    long long manhattanDistance(Point p1, Point p2) const;
    long long calculateClusterWireLength(Point center, const std::vector<Node*>& c);
    Point calculateBufferPosition(const std::vector<Node*>& cluster);
    void calculateFinalMetrics();
    void calculateArrivalTimes(Node* node, long long current_time);
    void populateLevelMap(Node* node);
    void cleanup();
};

#endif // CBI_H
```

cbi.cpp (Implementation File)

This file provides the logic for all the functions declared in `cbi.h`.

**Method 2: Simulated Annealing (SA-MMM) Experiment:** To explore the potential for further optimization, I implemented a hybrid algorithm that used the MMM tree as a starting point for Simulated Annealing. This experiment confirmed that while SA could find solutions with slightly smaller wire lengths, it did so at the expense of a significant and unacceptable increase in clock skew. This demonstrated a poor trade-off for the goals of this project.

for this project, I want to try Simulate Annealing method combines with methods Mean and medians, and the following is part of my code:"int main(int...

顯示思路

Of course. It looks like you're working on a fascinating EDA problem, combining a constructive heuristic (MMM) with a metaheuristic (Simulated Annealing). Your diagnosis is spot on: the issue is a classic state-management problem in Simulated Annealing.

When a "bad" move is rejected, the program's state isn't being correctly reverted. This causes your `current_cost` variable to become out of sync with the actual physical state of the clock tree, leading the algorithm to make incorrect decisions on subsequent iterations and "walk" towards a worse solution.

The core issue in your code is that while you correctly identify the need to `undoMove`, the surrounding logic doesn't robustly handle the state change and restoration.

### ## Corrected Simulated Annealing Implementation

Here is the corrected and more robust version of your `optimizeWithSimulatedAnnealing` function and its helpers. The key changes are:

- Modified `applyRandomMove`:** It now returns a `std::optional<MoveRecord>`. This is a modern and safer C++ pattern than using output parameters. It makes it impossible to forget to check if a move was actually made.
- Simplified Main Loop:** The main loop now correctly handles the optional return. The logic for accepting/rejecting moves is clarified to ensure state is never corrupted.
- Guaranteed State Sync:** The logic explicitly ensures that after a rejected move, the state is reverted (`undoMove`) and the metrics are recalculated (`calculateFinalMetrics`) so the next iteration starts from a consistent state.

Here are the code replacements. You will need to add `#include <optional>` to the top of your `.cpp` file.

1. cbi.h Header File

同問 Gemini

```
1. cbi.h Header File

The function signature for applyRandomMove needs to be updated. You'll also need to include the <optional> header.

C++

#ifdef CBI_H
#define CBI_H

#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <set>
#include <limits>
#include <random>
#include <optional> // <-- ADD THIS HEADER

// Forward-declare Node struct if it's used in MoveRecord
struct Node;
struct Point;

// A simple struct to record a move for easy undoing
struct MoveRecord {
    Node* node;
    Point original_pos;
};

// ... (Point, NodeType, Node structs remain the same) ...

class ClockTreeSynthesizer {
public:
    // ... (existing public methods) ...
    void optimizeWithSimulatedAnnealing();

private:
    // ... (existing private data members) ...
    std::mt19937 rng{std::random_device{}()};

    // ... (existing private methods) ...

    // THE CORRECTED HELPER FUNCTION SIGNATURES FOR SA
    std::optional<MoveRecord> applyRandomMove(); // <-- MODIFIED
};

#endif
```

Gemini



### Method 3: The RMST-based Method (Final Choice):

My final and best-performing algorithm is based on a **Rectilinear Minimum Spanning Tree (RMST)**. This approach was chosen because it directly optimizes for the project's primary cost metric: rectilinear wire length. The algorithm first constructs an RMST, which provides theoretically optimal wiring topology. It then partitions this

tree by making a "balanced cut" to ensure the resulting sub-groups are of similar size, which effectively maintains low clock skew.

ok, I got two questions, first, the first approach was added behind \* if (targets.size() <= maxFanout && directWireLength <= maxLength) \* case right? just... ✓

顯示思路

You've got it. Let's tackle both of your questions.

---

### ## 1. Correct Placement of the Optimization Code

Yes, your understanding is **perfectly correct**. You've placed the new optimization block in exactly the right spot.

The logical flow is now:

- Base Case:** Check if the current parent can drive all the targets directly. This is the cheapest option (no new buffer). If so, connect them and finish.
- Optimization Pre-check:** If the base case fails, check if a single new buffer can handle all the targets. This is the next cheapest option (only one new buffer). If so, create it, connect everything, and finish.
- Recursive Step:** If both of the above fail, it means the problem is too complex for a single driver. Only then do we proceed with the most expensive option: splitting the problem in two and creating multiple new buffers.

Your code snippet implements this priority system flawlessly. 🧠

---

### ## 2. How to Implement the RMST Method

Implementing a Rectilinear Minimum Spanning Tree (RMST) is a fantastic way to build a wire-length-optimal backbone for your clock tree. A great and straightforward way to do this is by using **Prim's algorithm**, adapted for a geometric context.

問問 Gemini

+ 工具

well, it doesn't work, the result are worse than the first algorithm, check it again, I think some algorithm should be re-consider again

顯示思路

You are absolutely right, and I apologize. My previous attempt to integrate timing-awareness was flawed. Forcing clusters based on a simple delay metric without considering their physical location was a mistake—it can create spatially scattered, inefficient clusters that worsen both wire length and skew. Thank you for catching that and pushing for a better solution.

Let's re-think this with a more robust and classic algorithm. The key is to find a balance: we need to group sinks that are physically close, but do it in a way that builds a balanced tree structure, which is **the most effective way to implicitly control clock skew**.

A powerful technique for this is **Recursive Bipartitioning**.

---

### ## The Recursive Bipartitioning Algorithm

Instead of greedily building one cluster at a time, this algorithm recursively splits the set of sinks into two balanced halves, creating a structure that looks much like a binary tree. This approach is superior because it ensures clusters are spatially compact and the resulting tree is inherently balanced, which directly tackles the skew problem.

Here is the rewritten `recursiveBuild` function implementing this logic.

```
C++  
  
void ClockTreeSynthesizer::recursiveBuild(Node* parent, std::vector<Node*> targets) {  
    // 1. BASE CASE: Unchanged. If the current parent can drive all targets  
    // directly while satisfying the constraints, the recursion stops for this branch  
    long long directWireLength = 0;  
    for(const auto& target : targets) {  
        directWireLength += manhattanDistance(parent->pos, target->pos);  
    }  
    if (targets.size() <= maxFanout && directWireLength <= maxLength) {  
        for (auto& target : targets) {  
            parent->children.push_back(target);  
        }  
        return;  
    }  
    // ... (RMST logic would go here) ...  
}
```

問問 Gemini

+ 工具

issues:

- Implicit Skew Control:** By always splitting the remaining sinks into two halves, the algorithm naturally creates a **balanced tree**. In a balanced tree, the path length (in terms of number of buffers) from the `SRC` to any sink is roughly the same. This structural balance is a powerful and robust way to minimize path delay variation, thus **reducing clock skew**.
- Guaranteed Spatial Locality:** The median split along a physical axis ensures that the sinks in each new cluster are physically close to each other. This avoids the previous algorithm's fatal flaw of grouping distant sinks, which guarantees that the wire length needed for each new buffer is kept low.
- Creates Natural Buffer Hierarchies:** This method inherently builds the multi-level buffer structures you mentioned. The `SRC` will drive two new buffers for the first split. Each of those buffers will, in turn, drive two more buffers for the next split, and so on. This continues until the remaining groups of sinks are small enough to be driven directly, resulting in a clean, hierarchical tree that effectively balances the load across the chip.

I'm curious that would the buffer insertion position coincides to be one of the sink? if it happens what would this code do?

顯示思路

That's a fantastic question that gets into a key detail of the implementation.

Yes, the calculated ideal position for a buffer can absolutely coincide with the position of an existing sink, and the code is specifically designed to handle this scenario.

---

### ## Can This Happen?

It's not only possible but quite likely to happen. The buffer's position is determined by finding the **median** of the x and y coordinates of all the sinks in its cluster.

By definition, the median of a set of numbers is one of the numbers in that set (or the average of the two middle ones). This means the calculated position for a buffer is very often

問問 Gemini

+ 工具

show me where to insert back to the above functions

顯示思路

Of course. Integrating the RMST logic involves replacing the simple geometric bipartitioning with a more intelligent partitioning guided by the RMST structure.

Here's where and how to fit the `buildRMST` function into your existing `recursiveBuild` function.

---

### ## Integration Strategy

The strategy is to use the RMST to make a smarter split. Instead of just cutting the group of sinks in half along an axis, we will:

- Build the RMST for the sinks.
- Find the **longest edge** in that optimal tree.
- "Cut" that longest edge. This will naturally partition the sinks into two spatially coherent groups.
- Create a buffer for each of these two new groups and recurse.

This replaces the old splitting logic with a new one that is based on the most efficient wiring topology.

---

### ## Where to Insert the Code

You will replace the entire section of code that calculates the bounding box and splits the sinks by the median.

Here is your `recursiveBuild` function. I've marked the exact section you need to replace.

```
C++  
  
void ClockTreeSynthesizer::recursiveBuild(Node* parent, std::vector<Node*> targets) {  
    // 1. Base case (parent drives targets) - REMAINS UNCHANGED  
    // ...  
    // 2. Bounding box calculation and median split (TO BE REPLACED BY buildRMST)  
    // ...  
}
```

問問 Gemini

+ 工具

---

### ## 3. Key Challenges and Debugging Process

The development process involved several key challenges:

- **Algorithm Evaluation:** A primary difficulty was rigorously evaluating the results from each algorithm to understand the inherent trade-offs between wire length and clock skew. The SA-MMM experiment was a perfect example of this, showing that a seemingly "better" wire length result is not superior if it compromises timing performance.
- **State Management in SA:** While not used in the final design, implementing the SA algorithm posed a significant debugging challenge. A subtle bug in the `applyRandomMove` function corrupted the program's state, making the `undoMove` function appear broken. Solving this with the AI's help was a critical learning experience in managing state within complex heuristic algorithms.

---

### ## 4. Conclusion

After implementing and evaluating multiple valid approaches, I chose the **RMST-based Partitioning method** for my final submission. This algorithm consistently produced the best results.

By starting from the Rectilinear Minimum Spanning Tree, it grounds the solution in a topology that is optimal for the project's wire length metric. By then using a balanced partitioning strategy, it maintains the lower-skew performance characteristic of a balanced tree. This synthesis of approaches proved to be the most effective strategy, delivering superior results when compared to the other methods I implemented. The interaction with the AI assistant was instrumental in rapidly prototyping these different options, allowing for a thorough, data-driven evaluation that led to this final design choice.

To sum up, through using the ai tool, this really reduce the time for debugging the codes , instead, I could use these time to focus on trying more ideas, especially for the complex recursive functions.