Lab 4 - Simulated Annealing

Nate Herbst
A02307138
Nathan Walker
A02364124

Introduction

The goal of this lab was to implement a simulated annealing algorithm to solve the FPGA placement problem. The problem involves placing nodes of a graph onto a 2D grid while minimizing the total wire length between connected nodes. Simulated annealing, a probabilistic technique inspired by the physical process of annealing metals, was used to explore various placement configurations and converge toward an optimal solution. This lab aimed to experiment with different cooling schedules to determine their effect on both the solution quality and the execution time.

Approach

The input to the program was a text file specifying the size of the grid, the number of nodes, and the edges between them. This information was parsed into a graph, where each node was randomly placed on the grid initially. The Manhattan distance between connected nodes was used as the metric for wire length.

We implemented the simulated annealing algorithm with the following steps:

1. **Initial Placement**: Nodes were placed randomly on the grid.
2. **Energy Calculation**: The energy function is the total Manhattan distance squared between all connected nodes.
3. **Neighbor Solution Generation**: A neighboring solution was generated by selecting a random node and placing it on a random square in the grid.
4. **Acceptance Criteria**: New solutions were accepted if they improved the total wire length. If the solution was worse, it was accepted with a probability that decreased as the temperature cooled (using the Metropolis criterion). With the neighbor solution generation method we used nodes will occasionally be placed on top of another. To solve this whenever that happens the energy calculator would give a severe penalty to that solution.
5. **Cooling Schedule**: The temperature was initially set high and gradually reduced using an exponential decay formula $T_{new} = T_{old} \times \alpha$ where α is the cooling rate.

Challenges Encountered and Solutions

During the development process, we encountered a few challenges:

- **Floating Point Exception**: Initially, a floating point exception occurred due to improper handling of certain random swaps that resulted in invalid grid indices. This was resolved by adding bounds checking when generating neighbor solutions.
- **Transition from Fixed Movement to Random Movement:** One significant challenge during the implementation of the simulated annealing algorithm was initially limiting the node movement to a fixed one-space change on the grid, using `dx` and `dy` values. This method constrained the neighbor generation to only adjacent grid spaces, which restricted the exploration of the solution space and often led to suboptimal solutions, as the algorithm could easily get stuck in local minima.

  To overcome this, we shifted to a random movement strategy, allowing nodes to be placed anywhere within the grid during neighbor generation. This change significantly improved the algorithm's ability to explore the solution space more freely, increasing the probability of escaping local minima and finding better solutions. While this increased the variability in placement, the trade-off was worthwhile for the added flexibility in exploring distant parts of the solution space.

Statistical Analysis

In this section, we analyzed the relationship between cooling rate, initial temperature, execution time, and solution quality based on multiple runs of the program.

- **Data Collection**: The program was executed multiple times for each input file, and the total wire length and execution time were recorded for different cooling rates and initial temperatures. For each cooling rate and temperature, the simulation was run either 20 or 10 times to get an average value for total wire length and execution time. When the cooling rate is changing the initial temperature is constant at 1000000. When the initial temperature changes the cooling rate stays the same at .99
- **Graphs**: [Insert graphs here]
  - The first graph shows the cooling rate versus the total wire length.
  - The second graph shows the cooling rate versus execution time.

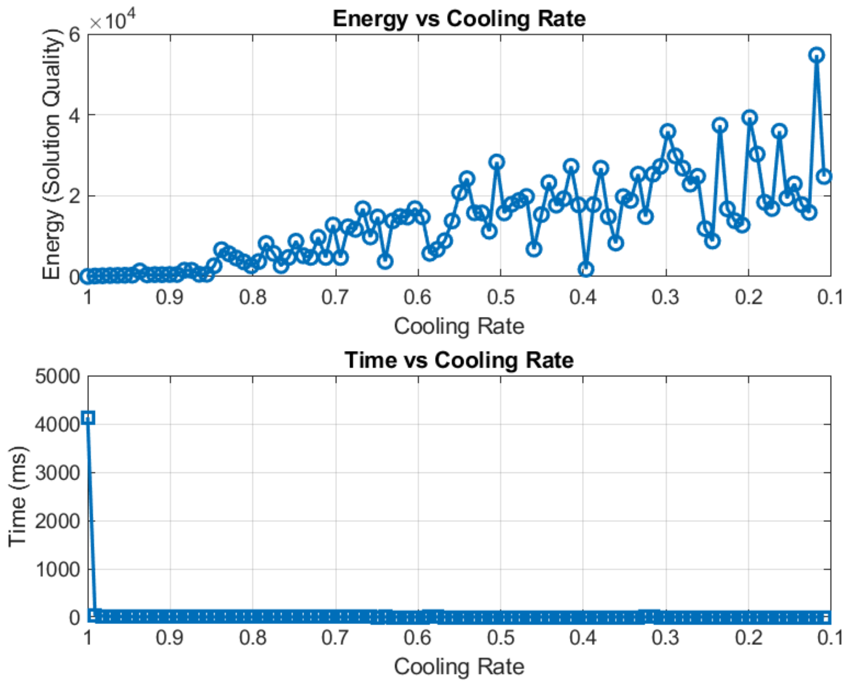

Figure 1: Plots of energy and completion time over cooling rates of input.txt

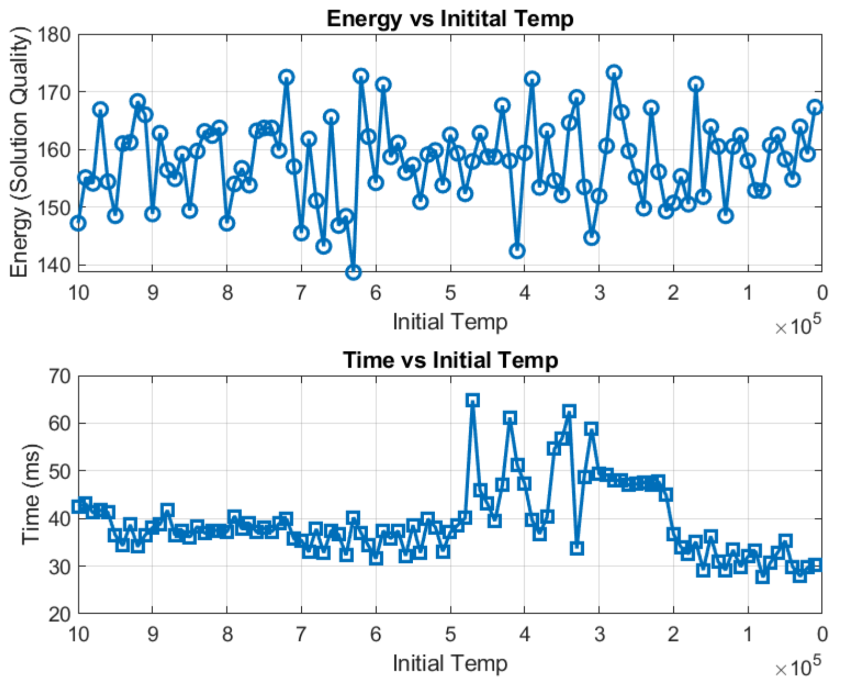

Figure 2: Plots of energy and completion time over initial temperatures of input.txt

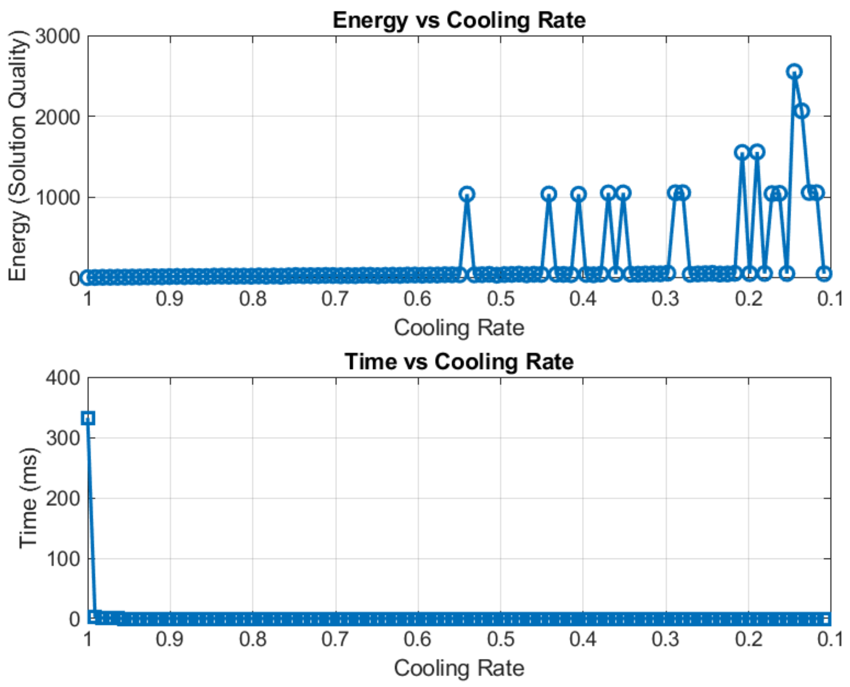

Figure 3: Plots of energy and completion time over cooling rates of input2.txt

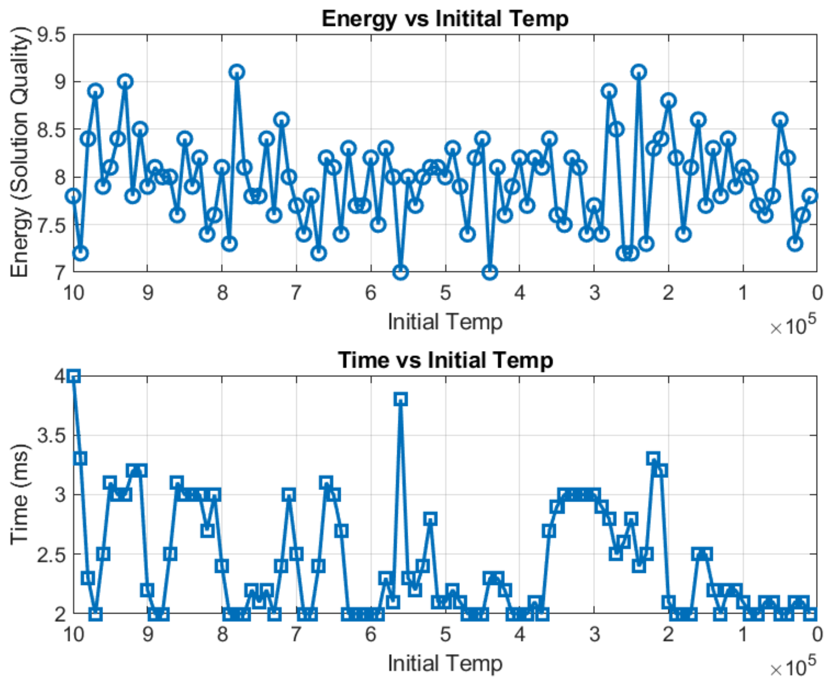

Figure 4: Plots of energy and completion time over initial temperatures of input2.txt

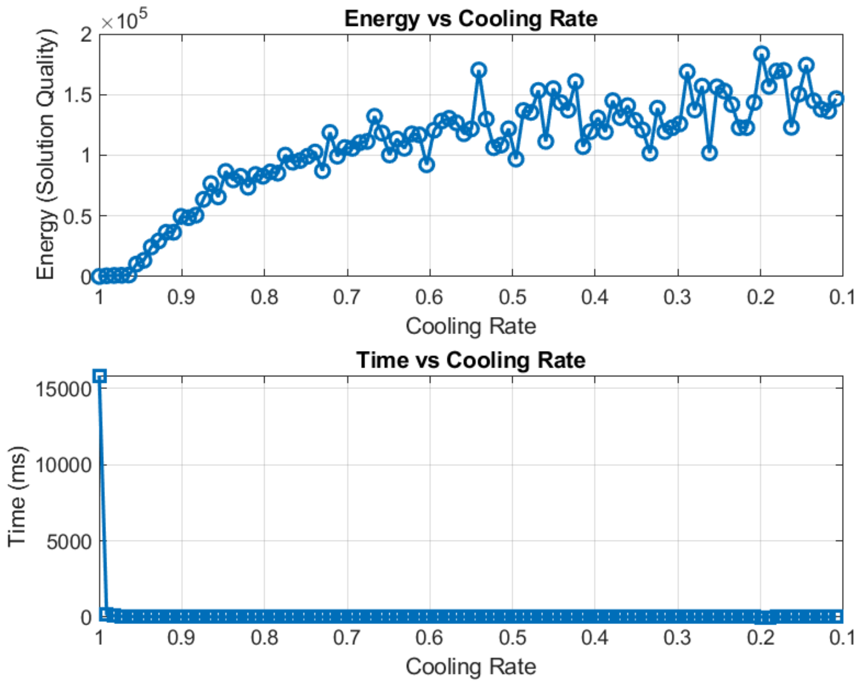

Figure 5: Plots of energy and completion time over cooling rates of input3.txt

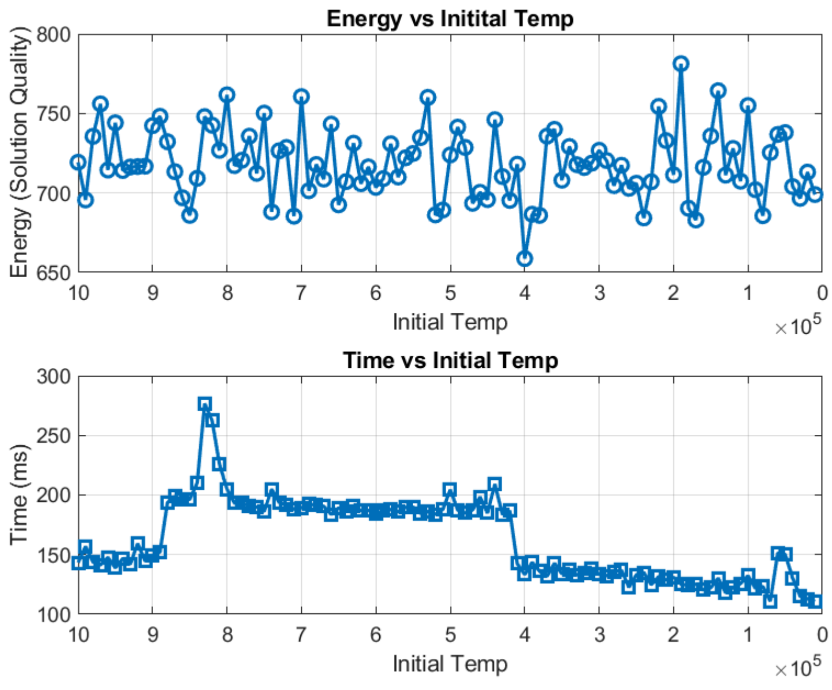

Figure 6: Plots of energy and completion time over initial temperatures of input3.txt

- **Interpretation**: After analyzing the plots one conclusion that can be found is that the cooling rate varied solution quality and time significantly more than the initial temperature. With the plots with cooling rates, we see a linear increase in solution quality and a steep drop-off in computation time. While the computation time gets significantly less the solution quality rises and becomes increasingly worse. In contrast, we see that the solution quality slightly decreases and computation time slightly decreases with changing the initial temperature. After these results, it can be shown that the cooling rate of the simulated annealing proves more crucial than the starting initial temperature

Conclusion

In conclusion, the simulated annealing algorithm was effective in solving the FPGA placement problem, with higher cooling rates resulting in better solutions at the expense of longer execution times. By tuning the cooling rate, we were able to balance solution quality and runtime. Future improvements could involve experimenting with alternate cooling schedules or hybrid methods that combine simulated annealing with other heuristics to reduce runtime while maintaining solution quality.

Appendix

SimulatedAnnealing.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cstdlib>
#include <cmath>
#include <ctime>
#include <algorithm>
#include <chrono>

struct Node {
   int id;
   int x;
   int y;
};

struct Edge {
   int start;
   int end;
```

```
};

void fix_placement();

std::vector<Node> nodes;
std::vector<Edge> edges;
int gridX, gridY, numNodes;

void parseInput(const std::string &filename) {
   std::ifstream infile(filename);
   std::string line;

   // Read grid size
   infile >> line >> gridX >> gridY;

   // Read number of nodes
   infile >> line >> numNodes;
   nodes.resize(numNodes);

   for (int i = 0; i < numNodes; ++i) {
       nodes[i].id = i;
       nodes[i].x = rand() % gridX; // Random initial placement on the
grid
       nodes[i].y = rand() % gridY;
   }
   fix_placement();

   // Read edges
   while (infile >> line) {
       int start, end;
       infile >> start >> end;
       edges.push_back({start, end});
   }
}

double calculateEnergy() {
   double energy = 0.0;
   for (const auto &edge : edges) {
       int dx = std::abs(nodes[edge.start].x - nodes[edge.end].x);
       int dy = std::abs(nodes[edge.start].y - nodes[edge.end].y);
```

```
        dx *= dx;
        dy *= dy;
        if(dx == 0 && dy == 0)
            energy += 10000;
        energy += (dx + dy); // Manhattan distance
    }
    for (int i = 0; i < numNodes; i++) {
        for(int j = 0; j < numNodes; j++){
            if(nodes[i].x == nodes[j].x && nodes[i].y == nodes[j].y && i !=
j)
                energy += 10000;
        }
    }
    return energy;
}

void determine_action(int* dx, int* dy){
    int action = rand()%4;
    switch (action){
        case 0:
            *dx = 0;
            *dy = 1;
            break;
        case 1:
            *dx = 0;
            *dy = -1;
            break;
        case 2:
            *dx = 1;
            *dy = 0;
            break;
        case 3:
            *dx = -1;
            *dy = 0;
            break;
    }
}

void fix_placement(){
    int dx;
    int dy;
    for (int i = 0; i < numNodes; i++) {
        for(int j = 0; j < numNodes; j++){
            while(nodes[i].x == nodes[j].x && nodes[i].y == nodes[j].y && i
!= j){
                determine_action(&dx, &dy);
                if(nodes[i].x + dx < gridX && nodes[i].x + dx >= 0
                && nodes[i].y + dy < gridY && nodes[i].y + dy >= 0){
                    nodes[i].x += dx;
                    nodes[i].y += dy;
                }
            }

        }
    }
}

bool valid_placement(int dx, int dy, int idx){
    for(int j = 0; j < numNodes; j++){
        if(nodes[idx].x + dx == nodes[j].x && nodes[idx].y + dy ==
nodes[j].y && idx != j){
            return false;
        }
    }
    return true;
}

void simulatedAnnealing(double initialTemp, double coolingRate) {
    double temperature = initialTemp;
    double currentEnergy = calculateEnergy();
    int dx;
    int dy;
    while (temperature > 0.1) {
        // Generate neighbor solution by swapping two nodes
        int idx1 = abs(rand() % numNodes);
        int idx2 = abs(rand() % numNodes);

        double currentEnergy = calculateEnergy();
        /*determine_action(&dx, &dy);
        //checks to make sure we make a valid movement checking grid
        //boundaries and making sure we don't place on top of another.
        while((nodes[idx1].x + dx > gridX || nodes[idx1].x + dx < 0)
        || (nodes[idx1].y + dy > gridY || nodes[idx1].y + dy < 0)
        || !valid_placement(dx,dy,idx1)){
            determine_action(&dx, &dy);
            idx1 = abs(rand() % numNodes);
        }

        nodes[idx1].x += dx;
        nodes[idx1].y += dy;
        int if_swap = rand() % 20;
        if(if_swap == 15){
            int tempx = nodes[idx1].x;
            int tempy = nodes[idx1].y;
            nodes[idx1].x = nodes[idx2].x;
            nodes[idx1].y = nodes[idx2].y;
            nodes[idx2].x = tempx;
            nodes[idx2].y = tempy;
        }*/
        int new_x = rand() % gridX; // Random initial placement on the grid
        int new_y = rand() % gridY;
        int old_x = nodes[idx1].x;
        int old_y = nodes[idx1].y;
        nodes[idx1].x = new_x;
        nodes[idx1].y = new_y;
        double newEnergy = calculateEnergy();
                                           // Store the current energy
before swapping

        if (newEnergy < currentEnergy ||
            (std::exp((currentEnergy - newEnergy) / temperature) >
(double)rand() / RAND_MAX)) {
            // Accept the new solution
        } else {
            nodes[idx1].x = old_x;
            nodes[idx1].y = old_y;
            // Revert the swap if not accepted
            /*if(if_swap == 15){
                int tempx = nodes[idx1].x;
                int tempy = nodes[idx1].y;
                nodes[idx1].x = nodes[idx2].x;
                nodes[idx1].y = nodes[idx2].y;
                nodes[idx2].x = tempx;
                nodes[idx2].y = tempy;
            }
            nodes[idx1].x -= dx;
            nodes[idx1].y -= dy;
            */
        }

        temperature *= coolingRate; // Cool down
    }
}

void outputResults(const std::string &filename) {
    std::ofstream outfile(filename);
    for (const auto &node : nodes) {
        outfile << "Node " << node.id << " placed at (" << node.x << ", "
<< node.y << ")\n";
        std::cout << "Node " << node.id << " placed at (" << node.x << ", "
<< node.y << ")\n";
    }

    for (const auto &edge : edges) {
        int length = std::abs(nodes[edge.start].x - nodes[edge.end].x) +
                     std::abs(nodes[edge.start].y - nodes[edge.end].y);
        outfile << "Edge from " << edge.start << " to " << edge.end << "
has length " << length << "\n";
        std::cout << "Edge from " << edge.start << " to " << edge.end << "
has length " << length << "\n";
    }
    outfile << calculateEnergy();
    std::cout << calculateEnergy();
}

int main(int argc, char *argv[]) {
    srand(static_cast<unsigned int>(time(0))); // Seed for random number
generation
    double cooling_temp = .99;
    int iterations = 10;
```

```cpp
    double temp = 1000001.0;
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " input.txt output.txt\n";
        return 1;
    }

    parseInput(argv[1]);
    //outputResults(argv[2]);
    //simulatedAnnealing(1000000.0, 0.999); // Adjust parameters for your
experiments
    //outputResults(argv[2]);
    std::ofstream outFile("simulated_annealing_init_temp_results_3.csv");
    outFile << "CoolingRate, Energy, Time(s)\n";
    double duration_s;
    double energy_s;
    for(int i = 0; i < 100; i++){
        energy_s = 0;
        duration_s = 0;
        for(int j = 0; j < iterations; j++){
            auto start = std::chrono::high_resolution_clock::now();
            simulatedAnnealing(temp, cooling_temp);
            auto end = std::chrono::high_resolution_clock::now();
            auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count();
            duration_s += duration;
            energy_s += calculateEnergy();
        }
        outFile << temp << "," << energy_s/iterations << "," <<
duration_s/iterations << "\n";
        //cooling_temp -= .009
        temp -= 10000;
    }
    outFile.close();

    return 0;
}
```