Lu Fei Liu – 14090154 April 25, 2021

CPEN 513 – Final Project

Genetics Algorithm for FPGA Placement and Graph Partitioning

Introduction

This report describes an implementation of the genetics algorithm for partitioning and placement in the FPGA CAD flow. The code is available at https://github.com/lucylufei/CPEN513-finalproject

Background

This final project is based on Assignment 2 and Assignment 3 of the course. As an alternative to simulated annealing for placement and branch and bound for partitioning, I've implemented a genetics algorithm to complete these tasks [1] [2]. Although genetics algorithms are a heuristic and do not guarantee the most optimal solution, they are helpful for quickly finding good solutions. However, genetics algorithms are not effective in fine-tuning, so the final results are not better than those found for Assignment 2 and Assignment 3.

General Implementation

The application follows the same format as Assignment 2 and Assignment 3 for comparative purposes. It first begins by reading the circuit parameters and netlist from the `.txt` file and generating an initial population of possible placements or partitions. Then, the application proceeds with the genetics algorithm for a number of generations before choosing the best member of the population as the solution.

The GUI has been maintained from Assignment 2 and Assignment 3 to graphically demonstrate the initial placement/partition and the final placement/partition results.

Placement

For placement, each gene is a vector of pin locations. Each element of the vector represents a pin, and the value of the element represents the cell location where the pin is to be placed. For example, $< 4, 10, 6, 3 >$ represents a circuit with 4 pins. The first pin is placed at location 4, second at location 10, and so on. The locations are encoded simply as $x \times num\_rows + y$, where (x, y) is the coordinate of the cell location.

During each generation of the algorithm, two parents are selected based on a fitness probability distribution. Fitness is determined by the half-perimeter cost used in Assignment 2 and uses the equation provided in the lecture slides. Then, a random index is selected to split each parent into two pieces. A single child is generated using the first part of the first parent and the second half of the second parent. Since pin placements cannot overlap, the second half of the child is copied with the condition that the location hasn't already been used. If the location has already been used in the first part of the child, the location for the pin will be copied from the first parent. If the locations for the pin in both parents have already been used, a random empty cell is

[1] T. N. Bui and B. R. Moon, "Genetic algorithm and graph partitioning," *IEEE Transactions on computers,* vol. 45, pp. 841--855, 1996.

[2] Z. Baruch, O. Cret and H. Giurgiu, "Genetic algorithm for FPGA placement," in *International Conference on Control Systems and Computer Science*, 1999.

selected. For example, given two parents of $< 3, 6, 2, 8, 10 >$ and $< 8, 4, 10, 6, 3 >$ with a random split chosen at index 2, the child would be $< 3, 6 \mid 10, 8, 1 >$ The first 2 pins before the split take their values from the first parent. The third pin is copied from the second parent. The fourth pin should be copied from the second parent, but the cell location 6 has already been used, so it is copied from the first parent. The last pin should also be copied from the second parent, but the cell location 3 of the second parent as well as cell location 10 of the first parent have both been used already. In this situation, a random empty location is selected, which helps create more mutations.

Once the child has been created, a random number of m mutations are applied. Each mutation replaces the location of a random pin with a random empty cell. Then, this child replaces the weakest member of the existing population and the generation is complete.

Partitioning

For partitioning, each gene is a vector of the partition that each pin belongs to. Each element of the vector represents a pin, and the binary value of the element represents whether the pin is in the left partition or the right partition. For example, $< 1, 0, 0, 1 >$ indicates that the first and last pin are in the right partition and the other two are in the left partition.

During each generation of the algorithm, two parents are selected, and two children are produced as described in the lecture slides. The children both go through mutations as described as well. To ensure a *legal* partition, the children go through a second round of mutation to ensure the pins are evenly distributed between the left and right partitions. This is done by selecting a random starting index and flipping each pin in the larger partition until both partitions are equal. These children then replace the two weakest members of the existing population and the generation is complete.

Parameters

The genetics algorithm is controlled by three main parameters: the number of iterations/generations (`n_iterations`), the size of the population (`population_size`), and the rate of mutation (`mutation_factor`). As the number of iterations and population size increases, the algorithm requires more time to complete, but it is more likely to arrive at a more optimal solution. Since the genetics algorithm is not suited to fine-tuning, very large values for these parameters come with diminishing returns. The rate of mutation controls how quickly better solutions are found. Without mutation, the population is not introduced to new solutions. However, too much mutation causes the algorithm to deteriorate to a random search process.

Results

Results are collected using the same benchmark circuits from Assignment 2 and Assignment 3 for ease of comparison. Table 1 shows the results for genetics algorithm placement, and Table 2 shows the results for genetics algorithm partitioning.

The genetics algorithm approach for placement is not nearly as effective as simulated annealing. It is easy for the genetics algorithm to get stuck in a local minimum. In simulated annealing, the number of undesirable moves accepted is initially high and decreases as the algorithm proceeds. In the genetics algorithm, this does not exist, so it is much easier to be stuck at a local minimum. However, the genetics algorithm executes quickly and does not require a carefully tuned annealing schedule. Simulated annealing is only effective when sufficient tuning has been completed, whereas the genetics algorithm only requires 3 parameters.

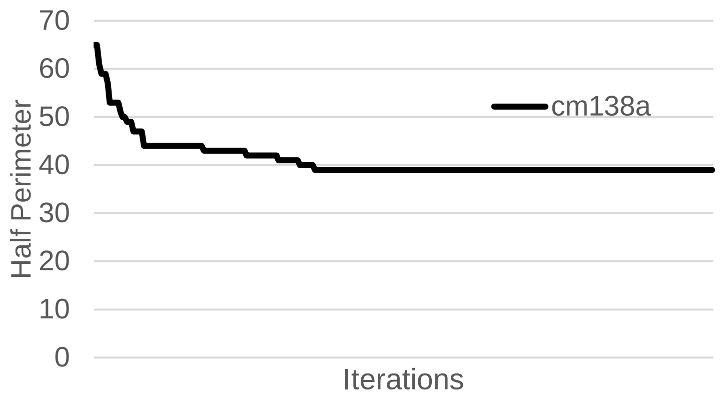

Figure 1 - Genetics Algorithm Progress Sample

Figure 1 demonstrates the progress of a genetics algorithm placer for a sample circuit. Unlike simulated annealing, the half perimeter only decreases, indicating that no undesirable moves are taken. This forces the algorithm into a local minimum. Then, it struggles to optimize further, as seen in later iterations. In order to improve the genetics algorithm implementation, more features must be added to avoid this problem.

The results presented in Table 1 is collected by running the genetics algorithm for approximately 1 hour, with a population of 100 and a mutation factor of 5.

Table 1 - Genetics Algorithm Placement Results

CIRCUIT	**FINAL COST (SIMULATED ANNEALING)**	**FINAL COST (GENETICS)**
alu2	995	1297
apex1	6332	14479
apex4	11472	33776
C880	1105	1763
cm138a	42	35
cm150a	67	66
cm151a	36	34
cm162a	77	83
cps	5507	14962
e64	2004	3854
paira	4327	18655
pairb	5715	30932
Average Cost	**3139.917**	**9994.667**

The same local minimum problem applies for genetics algorithm partitioning. Assignment 3 used the branch and bound method, which provides the guaranteed optimal cut size. Compared to this value, the genetics algorithm performs quite well in most cases. This is amplified by the significant runtime reduction. For example, the `cc` circuit required more than 4 hours to find the optimal cut size of 4 with branch and bound. The genetics algorithm finds a reasonable cut size of 6 within less than a minute. If performance is more important than the final cut size, then the genetics algorithm is the ideal choice.

However, this does not apply to smaller circuits. In branch and bound, once the optimal solution is determined, the algorithm will terminate. In the genetics algorithm, since we do not know what the optimal solution is, the algorithm continues for the set number of iterations even if it has already found the best solution it can find. This is unfortunate in smaller circuits such as `cm82a`, where the genetics algorithm takes more than 100 times longer to conclude with the same result. The genetics algorithm always requires a similar amount of time for all circuits, whereas branch and bound benefits from smaller circuits and vary significantly for larger circuits.

Table 2 - Genetics Algorithm Partitioning Results

CIRCUIT	**OPTIMAL CUT SIZE**	**RUNTIME (ASSIGNMENT 3)**	**CUT SIZE (GENETICS)**	**RUNTIME (GENETICS)**
cc	4	4:34:59.698272	6	0:00:29.814198
cm82a	1	0:00:00.002415	1	0:00:16.798954
cm138a	4	0:00:00.311096	4	0:00:19.316461
cm150a	6	1:04:30.836588	7	0:00:24.112514
cm162a	6	0:03:47.634649	8	0:00:23.981594
con1	4	0:00:00.092946	4	0:00:17.104239
twocm	1	6:30:19.544636	9	0:00:40.568345
ugly8	8	0:00:00.000553	8	0:00:16.905377
ugly16	16	0:00:00.001557	16	0:00:19.803598
z4ml	3	0:00:00.160984	3	0:00:17.247549

Application Guide

The program is run using either `placement.py` or `partition.py`. There is a GUI interface available, built with `tkinter`. Otherwise, a result file will be generated in the `logs/` folder. All settings are configured in `settings.py`.

In GUI mode, the circuit should be set in `settings.py`. Then, the user can set the initial partition or placement with the **Initialize** button and execute the genetics algorithm with the **Run Algorithm** button.

- **Initialize**: Get the initial population. This action can be repeated to observe various possible initial partitions or placements.
- **Run Algorithm**: Run the genetics algorithm. This produces the final partition and cut size or the final placement and half perimeter cost in the GUI.

The genetics algorithm can either be limited by a number of iterations to run, specified by `n_iterations`, or the user can set a time limit by entering minutes for the `time_limit` setting.

Table 3 outlines the contents of each Python file. All files relating to placement is located in the `placement/` folder and all files relating to partitioning is located in the `partition/` folder.

Table 3 - File Descriptions

FILE NAME	PURPOSE
`placement.py` `partition.py`	Main file containing GUI elements and runs the application
`netlist_parser.py`	Parses the circuit `.txt` into an appropriate format for the program
`util.py`	Contains useful functions shared between algorithms
`genetics.py`	Implementation of the genetics algorithm
`settings.py`	Contains settings for the program

Testing Procedure

Testing for this program was completed manually following the same methods in Assignment 2 and Assignment 3. The genetics algorithm results were also compared to the assignment results to ensure they are reasonable. There are some settings that also fixed partitions or placements as an input in order to observe them on the GUI. The program also maintains a debugging log file that can be manually checked and cross-referenced with the GUI to ensure the program is proceeding correctly.

A few extremely simple netlists were also designed as part of the testing procedure. These netlists are small enough to manually cross-check the cost function and ensure there are no cells missing or connections unaccounted for.

Otherwise, assertions are included throughout the program and exceptions are raised for unexpected results.

Conclusion

In conclusion, the genetics algorithm is an interesting approach to the placement and partitioning problems. It performs quite well but falls short of simulated annealing and branch and bound algorithms for nearly all circuits. The genetics algorithm is fast but often gets stuck in a local minimum. This can be improved by introducing more mutations that can trigger undesirable moves. A likely better alternative to explore would be using the genetics algorithms as an initial step, then fine-tuning with either simulated annealing or branch and bound.