ECE5960 Physical Design Algorithms Programming Assignment 2 Report

Cheng-Hsiang Chiu u1305418
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT-84112
{cheng-hsiang.chiu}@utah.edu

I. Introduction

In the assignmnet I implemented `Sequence Pair`[1] together with the optimization technique `Simulated Annealing` to solve the fixed-outline floorplanning problem.

II. Encountered Challenges

The first challenge I encounted was the incremental updates to the data structure. When a random moved was chosen, updating the all data structure to reflect this neighbor solution was less error-prone but extremely time-consuming.

The second challenge was the `fixed-outline` constraint. Because this is a fixed-outline floorplanning problem, I can not output a solution with a decent area but violate the constraint.

III. Ways to Overcome Challenges

To overcome the challeng of incremental updates, I identified the blocks that were affected by a chosen move and only updated the data structure of these blocks. For example, if two blocks were swapped in the positive sequence, I only updated the data structure of these two blocks and every block that located between them in the positive sequence. The same incremental updates applied to the remaining moves as well. In this way, I could avoid redundant data copy and reduce the runtime.

For the fiexed-outline constraint, I additionally added a penalty term to the cost function. The penalty term consisted of a penalty ratio and overshoot length. The penalty ratio was increased gradually such that solutions with overshoot outline happened less frequently than at the beginning. The overshoot length was a summation of the overshoot outline of the blocks whose outline exceeded the fixed-outline constraint. For example, the block `A` exceded the outline constraint. The overshoot lenght is equal to $(A.upper\_right\_x - outline\_x) + (A.upper\_right\_y - outline\_y)$.

IV. Availability

The whole implementation is available in the Github repository, link.

V. Build and Run Executable

There is a README in the uploaded compressed file and in the Github repository mentioned above. README provides very detailed information. In this section, I highlight the instructions to build and run the executable `fp`. Please make sure CMake and clang++ is installed.

To build the executable `fp`, use the following commands,

```
mkdir build
cd build
cmake ../
make
```

To run the executable with an input file, use the following command in folder `build`,

```
./fp [alpha] [block] [net] [output_file]
```

To verify the correctness, a python script will be copied to folder `build` automatically when building `fp`. In folder `build`, simply use the following command,

```
python3 checker.py [input] [output_file]
```

In addition to running `fp` with one input, I provide a script `run.sh` to run `fp`, measure the runtime and verify the correctness with all of the inputs. The script will be copied to folder `build` automatically when building `fp`. In folder `build`, simply use the following commands,

```
chmod 744 run.sh
./run.sh
```

VI. Results

The initial positive and negative seuqnces were random sequences. The executable was built using clang++ v10.2 with -std=c++17 and -O3 on a Linux machine with Intel i7-9700K 8 Cores at 3.6 GHz and 32 GB RA. All results were obtained in a single run.

Table I shows the total cost and runtime for each input.

References

[1] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "Vlsi module placement based on rectangle-packing by the sequence-pair," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1996.

Input	Total Cost	Runtime (second)
1	360500	0.526
2	360175	5.898
3	101947747	68.82
ami33	651367.5	45.452
ami49	19459090	58.779
apte	24114125.5	5.638
hp	4858805	5.323
xerox	10606274	1.699

TABLE I: Total cost and runtime of each input.