Master's Thesis

Greedy-K: A New Local Optimization Algorithm for Solving 1D Circuit Placement Problems

회로의 1차원 배치문제 해결을 위한 새로운 지역최적화 알고리즘

February 2023

Graduate School of Engineering
Seoul National University
Department of Computer Science and Engineering

Hyung Dal Kwon

Greedy−K: A New Local Optimization Algorithm for Solving 1D Circuit Placement Problems

Jae Jin Lee

Submitting a master's thesis of
Engineering

February 2023

Graduate School of Engineering
Seoul National University
Department of Computer Science and Engineering

Hyung Dal Kwon

Confirming the master's thesis written by
Hyung Dal Kwon
February 2023

Chair _____(Seal)
Vice Chair _____(Seal)
Examiner _____(Seal)

Abstract

Area reduction is one of the most critical objectives in semiconductor design since it improves profitability due to increasing net die per wafer. Although there exist various commercial tools, memory design requires the full custom design flow to reduce the area because the place and route (P&R) functionality in the tools are not effective in the dram design flow. Furthermore, one-dimensional (1D) layout is indispensable due to the presence of peripheral regions. Inspired by the above, we propose a new framework to minimize the wire length in the standard cell's 1D layout. The framework consists of the heuristic algorithm, which efficiently places standard cells to minimize the overall wire length of a 1D unit block composed of multiple standard cells and a *Clustering* algorithm. Through the cooperation of three algorithms, it obtains the 26.6% improved total wire length on 502 units consisting of 3 to 98 standard cells designed by human experts.

Keyword: 1D placement, permutation, wire length, optimization, clustering
Student Number: 2021-27781

Table of Contents

Table of Figures

Table of Algorithms

Table of Tables

Chapter 1. Introduction

In general, the area utilization of ASIC chips is difficult to exceed 70%, but the memory chip is more than 90%. In ASIC design flow, the standard cell placement process is performed by commercial tools that use force-directed methods to alleviate the probability of design rule constraints (DRC) violations, such as maximum fanout and maximum transition time, and to prevent routing congestion. However, the gap between cells induced by the force-directed method aggravates the chip area's utilization. On the other hand, in the memory design process, all cells, including standard cells and customized cells, are placed by human experts who use know-how based on their experience. They usually abut standard cells in a 1D manner according to the functions and stack them to generate 2D designs. The reasons for the difference between the two design flows come from that the production cost of relatively high-selling memory is more important than the production cost of ASIC chips and the clock frequency of memory is several times slower than that of ASIC chips.

Abutment of standard cells looks like a straightforward and efficient strategy to reduce the area of chips but reducing cell distance increases the possibility of routing congestion. The congestion may result in undesirable phenomena such as timing violation due to longer wire lengths to avoid congestion and crosstalk. To address the issue, efficient cell placement is necessary to reduce the semiconductor wire's length. The optimized wire length can improve SI (Signal Integrity) and PI (Power Integrity) characteristics, ease of routing, maximum transition time reduction, and higher frequency.

And as mentioned in [3] and [9], the placement for wire length minimization is a well-known NP-hard problem and a kind of combinatorial optimization problem. Various algorithms, such as genetic algorithms and simulation annealing (SA, [3]), are widely used to solve this problem by combining with the heuristic search

algorithms. However, heuristic algorithms used in SA to search local optima in commercial tools have yet to be disclosed. Although [1] and [4] are one of the most powerful heuristic search algorithms to search local optima in Max-Cut problem and Traveling Salesman Problem (TSP), respectively, they cannot be applied directly to the 1D placement problem due to the difference of problems.

In this paper, we first introduce the 1D placement problem and propose three novel algorithms, a heuristic optimization algorithm and a *Clustering* algorithm, which efficiently place standard cells to minimize the total wire length of a 1D unit block consisting of multiple standard cells.

Chapter 2. Related Work

2D placement of macro cells such as SRAM, and analog blocks has been studied for a long time. In [8], the sequence pair representation and min-cost flow are used to minimize total wire length. But there is no information of the processing time for many cells and just focusing on optimizing the wire lengths in the post-floorplanning phase. The reinforcement learning (RL) algorithm is used for placement in [6]. However, they also focused only on the floor planning level and only used force direction methods to place standard cells.

In [7], the author introduces a new *Clustering* method using connectivity and distance-based cell grouping to place standard cells efficiently. Limiting the number of cells in the cluster to 2k or 4k is a decision that does not consider the design, so it is difficult to obtain optimal results, and it is hard to produce good results in a short time due to a large number of cells.

As mentioned in the previous section, the placement problem is an NP-hard problem. 6 NP-hard problems, including TSP(Traveling Salesman Problem), are introduced in [5] and various reinforcement learning methods were used to attempt to solve the problem. However, as seen in the paper, the most powerful algorithm known to solve the TSP problem is the LKH(Lin–Kernighan–Helsgaun) introduced in [2]. Therefore, it can be seen that designing an efficient heuristic algorithm is valuable in solving 1D placement problems

Chapter 3. Contributions

To the best of our knowledge, this is the first work to solve 1D circuit placement problems with the following contributions:

- A new heuristic algorithm, *Greedy-K*, solves 1D placement problems quickly and efficiently compared to the genetic algorithm.
- A flexible *Clustering* algorithm for 1D placement problems achieves better results in the units with a large number of cells.
- We identified that *Cell Flipping* and *DFS&BFS Initialization* is effective in 1D placement problems.
- We compared the results of the above algorithms with 502 units consisting of 3 to 98 standard cells created by human experts.

Chapter 4. Problem Definition

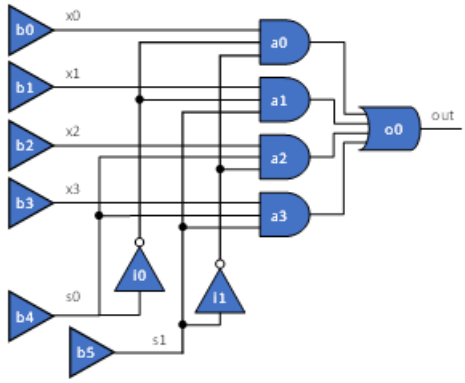


Figure 1 An example of a circuit, 4:1 Multiplexer (Mux)

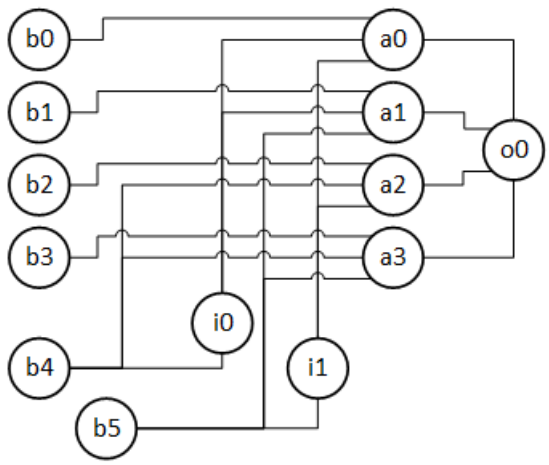


Figure 2 A graph representation of 4:1 Mux

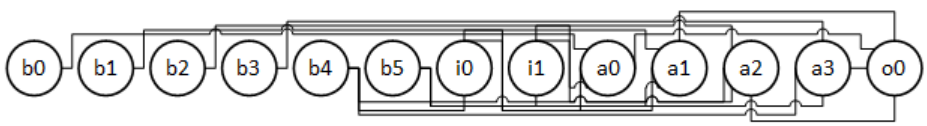


Figure 3 1D placement of 4:1 Mux

1D placement problem is a permutation problem. To minimize the total wire length (TWL) of the target unit block, you must first convert a netlist containing the connection information of the circuit and wire into a graph, such as Fig. 1 and Fig. 2. Then, we can

assign a unique number to each cell converted to a node in the graph and list the number of nodes as a one-dimensional vector to determine the arrangement order of each circuit as shown in Fig. 3. Since information about the height and width of each cell and its port can be obtained from the physical library in the Process Design Kit (PDK), the total wire length can be calculated given the order of cells and wire connection information. By randomly changing the order of numbers and obtaining the sum of the wire lengths, the sum of the wire lengths reduced more than before can be obtained. However, since it is difficult to obtain a good result quickly from a randomly placed permutation, a local optimal solution close to the random permutation can be quickly obtained in a limited time by applying a heuristic algorithm called *Local Optimization* algorithm.

Chapter 5. Methodology

In general, algorithms such as GA and SA are used to improve the solution in candidates generated by random numbers to solve the NP-hard problem. We intend to use GA as a control for *Local Optimization* and as an assistant for *Local Optimization*.

5.1. Structure of Genetic Algorithm

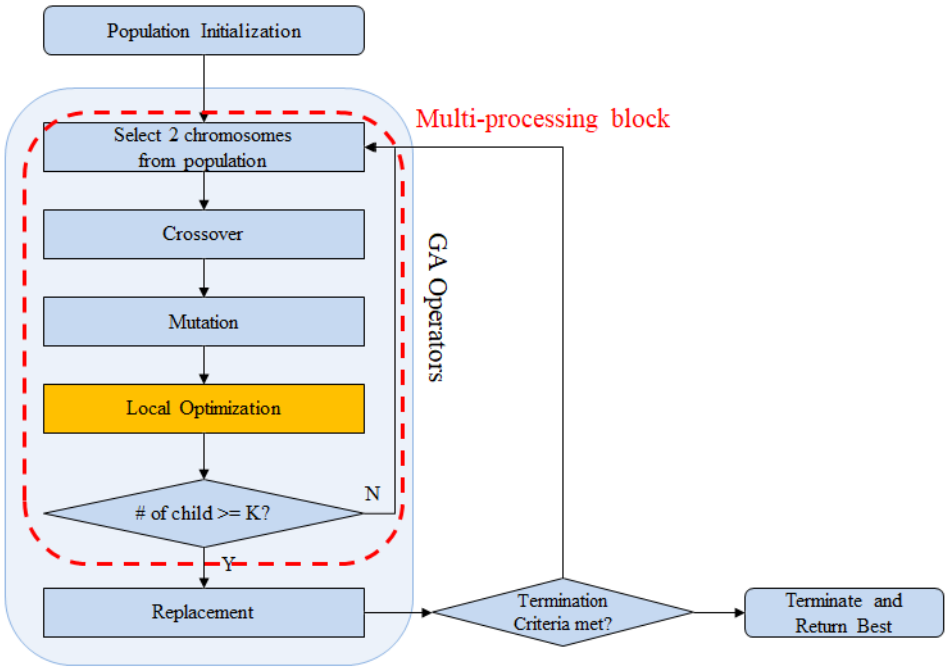


Figure 4 Structure of Genetic Algorithm

The structure of GA is in Fig. **오류! 참조 원본을 찾을 수 없습니다.**4. A number of solution candidates are generated with a random number in Population Initialization step. And, two candidates called parents are selected to create a better solution in Selection step. In Crossover step, two parents are mixed to generate an offspring. This step uses either Partially Matched Crossover or Ordered Crossover, but there is no performance difference. In Mutation step, the two random numbers swap positions, and the number of times the two are chosen can be adjusted, and *Local Optimization* algorithms can be applied to the offspring after Mutation step. Changing the number of runs from Selection to Mutation can determine the number of offspring, and the ratio of offspring to *Greedy−K* population is usually set between 10% and 20%. After generating offspring, Among the candidates in the population, candidates whose quality is lower than that of the offspring are exchanged with the offspring, thereby increasing the

quality of the entire population. The procedure described so far is referred to as the first generation, and a locally optimal solution can be found by repeatedly executing the generation.

5.2 MultiStart

Algorithm 1 MultiStart

```
bestSolution = createCandidate()
stop = False
while stop ≠ True do
    candidate=createCandidate()
    LocalOptimization(candidate)
    if candidate.quality() > bestSolution.quality() then
        bestSolution = candidate
    else if candidate.quality() == bestSolution.quality() then
        stop = True
    end if
end while
return bestSolution
```

Algorithm 1 MultiStart

MultiStart means that the *Local Optimization* algorithm is applied to randomly generated candidates without GA or SA as shown in Algorithm 1. We evaluated the performance of *Local Optimization* in Section 6.

5.3 Greedy-K: New Local Optimization Algorithm

Algorithm 2 Greedy-K

```
1: Get an input candidate and k, the maximum number of cells to move
2: run = True
3: while run do
4:     Prev_value = Calculate_TWL_of_candidate()
5:     Weight_dict = Calculate_dictionary_of_pairs() # {node:(sum of left wire length, sum of right wire length)} of each cell
6:     Weight_list.sort() # Sorts each cell in descending order based on the absolute value of the two values subtracted.
7:     i=0
8:     for each node in Weight_dict do
9:         if i == k then
10:            break
11:        end if
12:        TWL_history=Evaluate_TWL_at_all_position(Prev_value, node)
13:        Cur_value = Move_best_position(TWL_history)
14:        if Cur_value == Prev_value then
15:            run = False
16:            break
17:        end if
18:        Prev_value = Cur_value
19:        i=i+1
20:    end for
21: end while
22: return best solution
```

Algorithm 2 Greedy-K

The *Local Optimization* algorithm is essential to find optimal solutions in spaces with near-infinite search ranges quickly. We designed a novel *Local Optimization* algorithm to solve the 1D placement problem quickly and efficiently.

As mentioned on line 3 in Algorithm 2, we need to determine the order of cells to move, and for this, we use the method shown in Fig. 5. The sum of the lengths of the left and right wires connected

to the cells is calculated, and the movement order of each cell is determined by arranging them in descending order based on the value obtained by subtracting two numbers. For simplicity, the weight values in Fig. 5 are shown as values without direction. Between the experimental result of determining the movement order by subtracting the sum of the lengths of both wires and the experimental result of adding the sum of the lengths of both wires, the former result was better.

In order to increase the efficiency of the operation, a parameter ' k' as shown in Algorithm 2 is provided to control the number of cells to be moved so that the results sorted by wire length can be used multiple times. If you want to use only the latest sorting results, you can set k to 1. Through experiments, it was confirmed that an optimal value can be found by setting only the parameter k at 40% of the permutation length for a relatively short candidate with a permutation length of 50 or less.

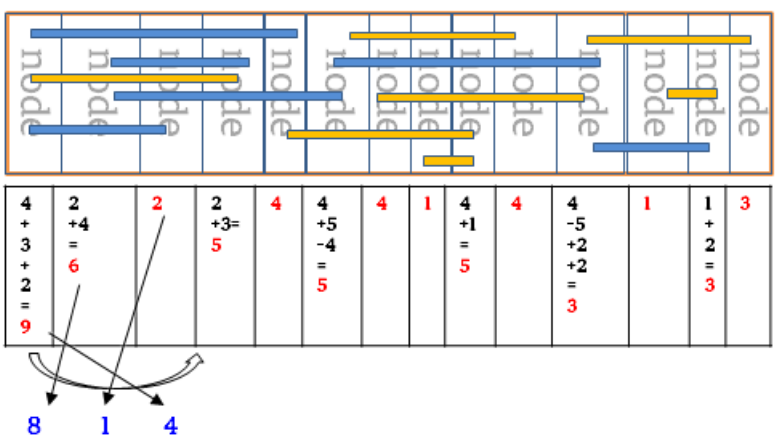


Figure 5 An example of calculating the weight of cells

5.4 Clustering for 1D Placement Problems

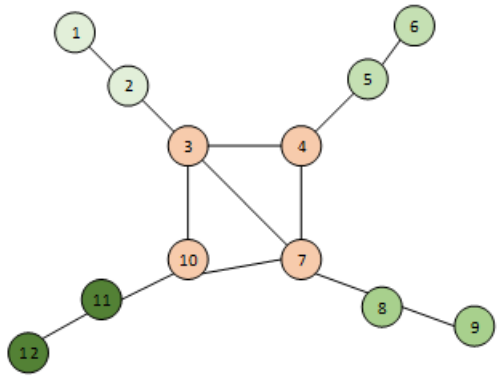


Figure 6 Example of color-coded clusters

Node	1	2	3	4	5	6	7	8	9	10	11	12
Cluster	1	1	2	2	3	3	2	4	4	2	5	5
fixed	1	1	0	0	1	1	0	1	1	0	1	1

Relocation

Node	1	2	3	4	7	10	5	6	8	9	11	12
Cluster	1	1	2	2	2	2	3	3	4	4	5	5
fixed	1	1	0	0	0	0	1	1	1	1	1	1

Local optimizations for clusters

Node	1		3				5		8		11	
Cluster	1	1	2	2	2	2	3	3	4	4	5	5

Figure 7 Clustering viewed at the cell level

As the number of cells increases, the processing time increases rapidly due to the exponentially increasing search space. To solve this problem, we combine a number of closely connected cells into one cluster, reducing the size of the search space so that the

optimal solution can be found quickly. Nodes with a degree of 2 or less of nodes are unlikely to have a negative effect on the sum of the total wire length even if they are grouped into one cluster and moved together, so we decided to combine them into one cluster.

In general, nodes with a node degree of 2 or less appear in the leaf node of the graph, sometimes inside the graph. In addition, the algorithm was designed so that complex nodes with node degrees of 3 or more can be grouped into clusters to see what effect it will have if it is optimized. Therefore, the criterion of our *Clustering* algorithm is the node degree. In Fig. 6 and 7, which shows an example of clustering, there are four green clusters consisting of nodes with a degree of 2 or less and one orange cluster consisting of nodes with a degree of 3 or more. The *Local Optimization* algorithm is applied separately to each cluster, and after applying the *Local Optimization* algorithm to the cluster, the GA algorithm and the *Local Optimization* algorithm are applied using the cluster and nodes not included in the cluster. Since the user can adjust the upper and lower limits of the node degree, which is the criterion for *Clustering* as shown in Algorithm 3, it is possible to create and experiment with various clusters.

Algorithm 3 Clustering

```
selectedNode = []
for each node in graph do
    if node.Degree is between lower and upper limit then
        selectedNode.append(node)
    end if
end for
#Apply Breadth-First Search for each selected node
clusters = []
for each node in selectedNode do
    visited_nodes = []
    queue = []
    visited_nodes.append(node)
    queue.append(node)
    while queue is not empty do
        cur_node = queue.pop()
        for adjacent_node of cur_node do
            if adjacent_node not in visited_nodes and adjacent_node.Degree is between lower and upper limit then
                visited_node.append(adjacent_node)
                queue.append(adjacent_node)
            end if
        end for
    end while
    clusters.append(visited_nodes)
    selectedNode.remove(visited_nodes)
end for
return clusters
```

Algorithm 3 Clustering according to the degree of nodes

5.5 Cell Flipping

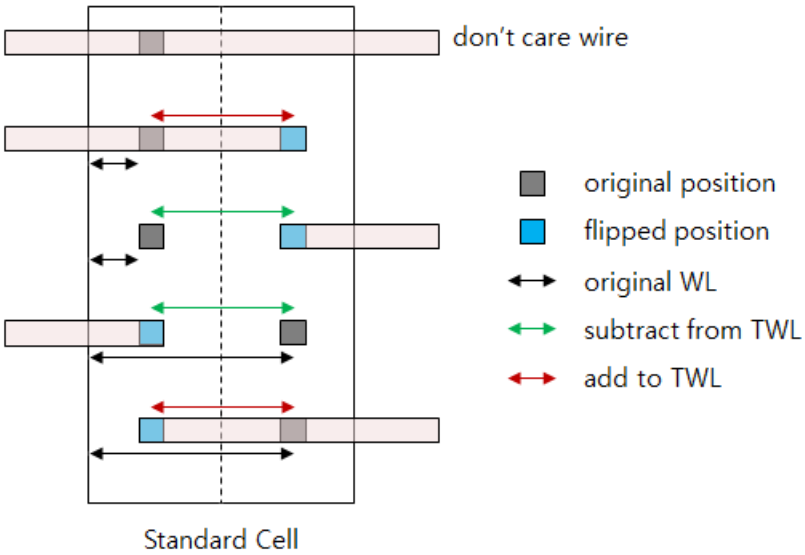
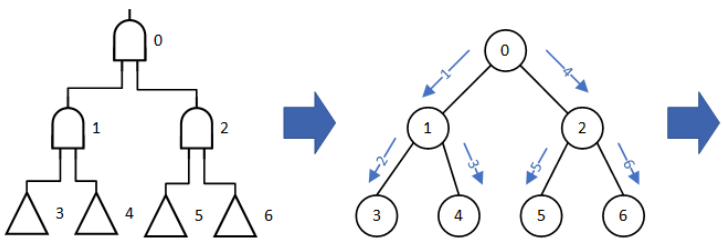


Figure 8 Wire length calculation methods of Cell Flipping

The standard cell can be reversed horizontally along the vertical axis to reduce the length of the wire connected to the cell. We call this function as *Cell Flipping* and apply it after cell placement. Prior to this algorithm, flips were determined by experts but were automated by *Cell Flipping*. In Fig. 8, the grey color square indicates the contact of a port before the flip, and the blue square box indicates the contact of a port after the flip. The wire length of the cell can be calculated by comparing the port positions before and after the flip.

5.7 DFS&BFS Initialization



Root	DFS Ordered Permuations						
0	0	1	3	4	2	5	6
1	1	3	4	0	2	5	6
2	2	0	1	3	4	5	6
3	3	1	4	0	2	5	6
4	4	1	3	0	2	5	6
5	5	2	0	1	3	4	6
6	6	2	0	1	3	4	5

Figure 9 An example of DFS Initialization with simple circuit

DFS&BFS Initialization algorithm is one of the well-known algorithms that use the topology of a circuit to generate initial candidates as shown in Fig. 9. Compared to random initialization, there is an effect of improving the quality of initial candidates. If N cells are candidates, *DFS&BFS Initialization* algorithm can generate N candidates from DFS and BFS, respectively, using all elements as the root. We have limited the number of candidates generated in DFS and BFS respectively to not exceed one-third of the population for the diversity of the candidates. For example, if the number of candidates in a population is 30, DFS will generate 10 candidates, BFS will generate another 10 candidates, and the rest will be randomly generated.

Chapter 6 Experimental Results

We validate the algorithm introduced in Section 5 using 502 unit blocks consisting of 3 to 98 standard cells.

6.1 Test Environment and Condition

Name	CPU	# Sockets	DRAM(GB)
A	AMD EPYC 7543, 32core 64thread	2	1024
B	Intel Xeon Gold 6142, 8core 16thread	1	128
C	Intel Xeon E5-2620 v4, 16core 32thread	2	256

Table 1 Server specification used for validation

Option Name	Abbreviation	Default Value	Description
population	p	128	Number of candidates
selection	s	128	Number of selection of 2 parents to make an offspring
iter	i	3000	Number of iterations(generations in Genetic Algorithm)
k-of-roulettewheel	kr	4	Control value of the maximum probability-to-minimum probability ratio in the selection operation
cutpoint-rate	c	0.6	Number of cutpoints in PMX crossover and is determined by the ratio of cutpoints to the length of chromosome
mutation-rate	m	0.15	The ratio of cells to be swapped on chromosome
genitor-replacement-rate	grr	0.2	Random replacement ratio in genitor replacement operation
dfsbfs-initilization	dbinit	True	Enable DFS&BFS Initialization
flip	f	True	Enable Cell Flipping
clustering	c	False	Enable Clustering
clustering-method	cm	None	Specify clustering method. You can choose one or more between leaf and low-degree

Table 2 Hyperparameters of Genetic Algorithm

For validation, we used three types of servers as shown in Table 1 and python3.9 to create a framework that can operate the algorithm introduced in Section 5. Table 2 shows the hyperparameters used in the framework. If some test cases use the value different from the default value, it will be described in this paper.

The data set has 502 unit blocks that consist of 3 to 98 cells. 50% of units only have under 8 cells and 80% of units have the number of cells under 40. The program termination condition is when the TWL difference between the best candidate and the worst candidate is less than 1e−4, and the sum of the number of best candidates and the worst candidates exceeds the population or the number of best candidates exceeds 70% of the population. The unit of length is micrometer and the default unit of time is second.

6.2 Performance of DFS&BFS Initialization

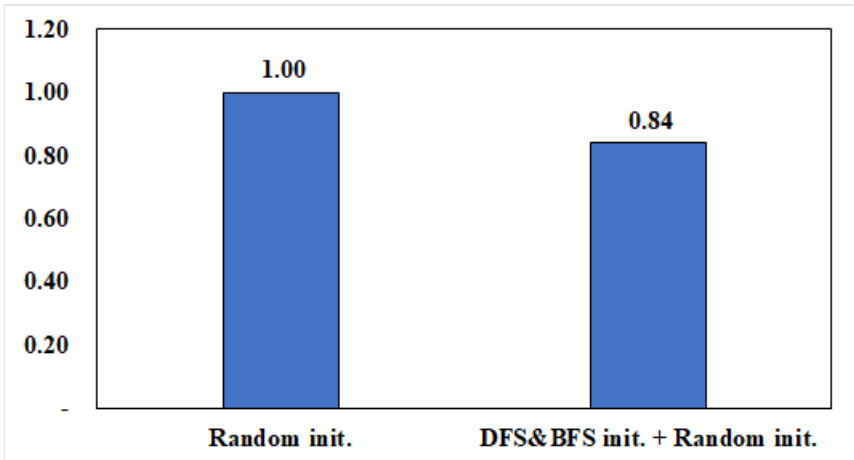


Figure 10 Performance comparison between DFS&BFS initialization and the random initialization

We evaluated the *DFS&BFS Initialization* algorithm in 124 units of 3 to 96 cells. As shown in Fig. 10, 16% of TWL improvement was confirmed by applying the *DFS&BFS Initialization* algorithm, and it became an algorithm basically applied in all tests of this paper.

6.3 Performance of Cell Flipping

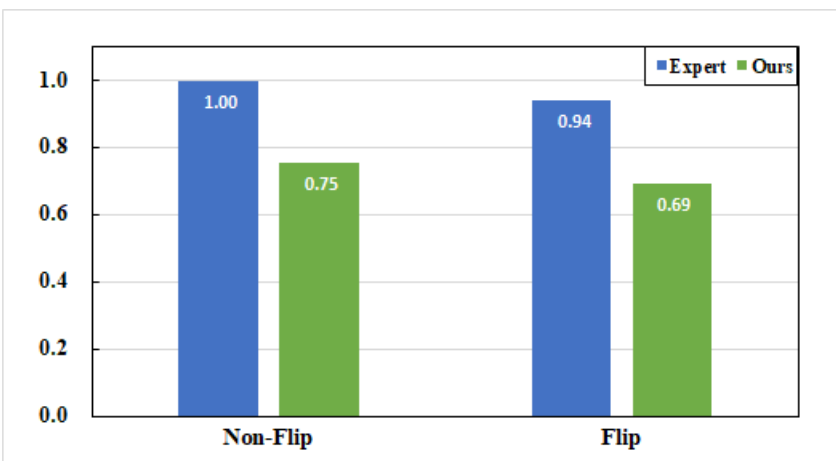


Figure 11 Normalized TWL comparison of expert and Greedy-K according to the flip operation using 502 units

As shown in Fig. 11, the *Cell Flipping* algorithm improves TWL by 6% compared to the case before applying *Cell Flipping* in both human-designed results and the algorithm's design results. In this test, *Cell Flipping* was applied to all 502 units with the *DFS&BFS Initialization* algorithm, and it was also decided to apply *Cell Flipping* to all test cases in this paper.

6.4 GA vs. Greedy-K vs. GA+Greedy-K

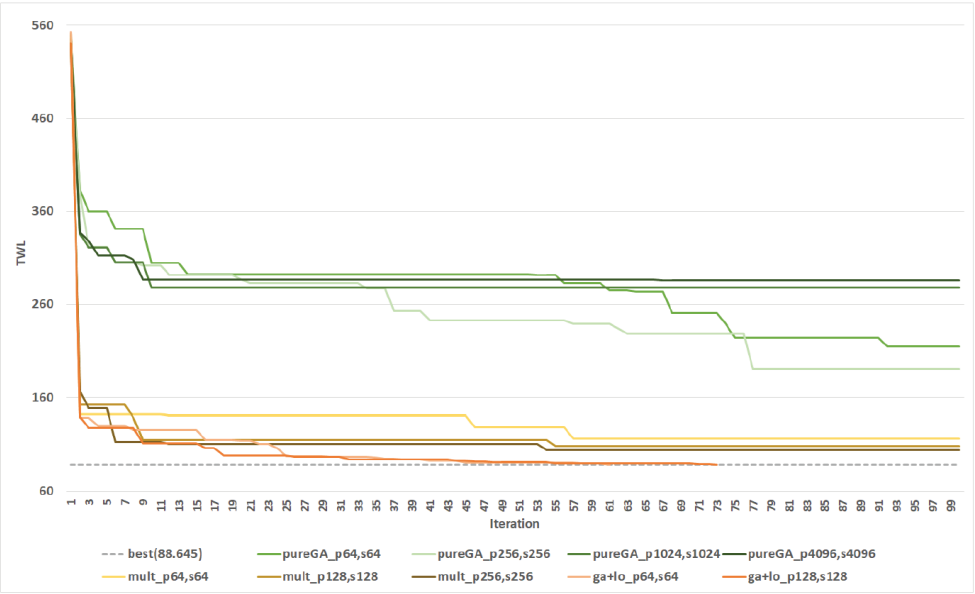


Figure 12 Performance comparison, PureGA vs. MultiStart(Greedy-K) vs. GA+ Greedy-K. Optimal value is 88.64.

In Fig. 12, ga means pure GA to which the Greedy-K algorithm is not applied, and mult means MultiStart and the Greedy-K, Local Optimization, algorithm applied to a population without GA operation and ga+lo means that GA and the Greedy-K algorithm are applied to the population together. And, the numbers after pureGA, mult, and ga+lo indicate the number of populations and the number of offspring generated after one iteration as shown in Table 2. For example, mult_p128,s128 means that the total number of populations used in MultiStart is 128, and the entire population is

subject to improvement for each iteration. For the evaluation of Fig. 12, we used a circuit that has 29 cells and 36 wires. As shown in the Fig. 12, when 100 iterations are performed, pure GA can see slow improvement in quality, and when opting GA and Greedy-K are applied together, it can be seen that the local optimal solution is reached the fastest. From this, it can be seen that GA prevents Greedy-K from quickly falling into the local optima and allows a better solution to be found. Pure GA can find the optimal solution when the cell length of the candidate is short, but through the inverter chain test, the inverter connected serially, we found that if the cell length of the candidate exceeds 50, GA never finds the optimal solution. However, a chain test using a circuit in which the inverter is connected in series found that if the candidate cell length exceeds 50, GA does not find an optimal solution. And, the difference in execution time between pure GA and GA+Greedy-K is up to 2,500x. Note that all tests are applied with both GA and Greedy-K unless otherwise noted.

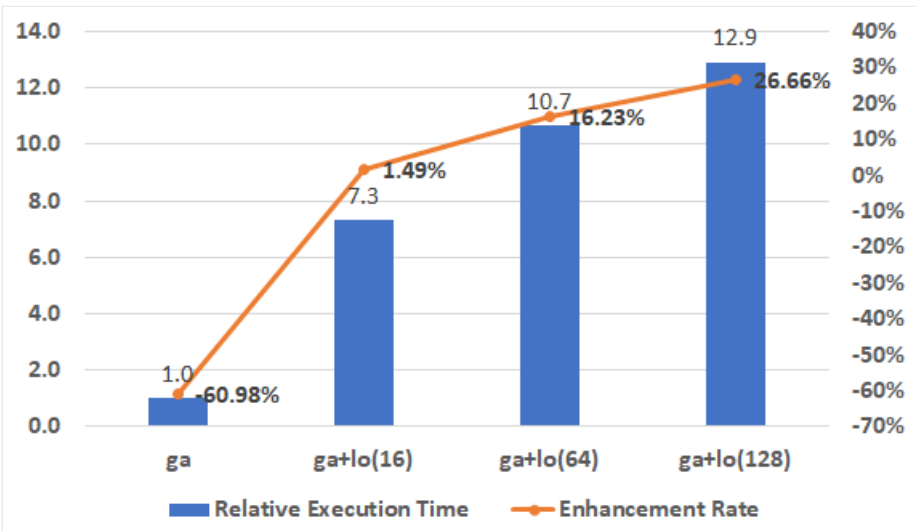


Figure 13 Performance of GA+ Greedy-K according to the number of population and selection

Typically, in the selection step of GA, 10 40% of the population is selected to produce offspring, but in our evaluation, the same

number of selections as the population is evaluated to yield better results. In the ga+lo(16) in Fig. 13, the number of populations was set to 64 and the number of selections to 16, and the number of selections and the number of the population were the same in the remaining test cases. And we can know that the execution time increases with the number of populations and selections.

6.5 Human Experts vs. Cell Flipping vs. Clustering

Test Case	Left is better	Even	Right is better	Total
Flip vs. Experts	281	219	4	502
Leaf vs. Flip	11	376	115	502
Leaf+Low-Degree vs. Flip	12	347	143	502

Table 3 Performance comparison, Human expert vs. Flip, Leaf node clustering vs. Flip, Leaf + Low-degree clustering vs. Flip in 502 units test. Leaf=Flip+ Leaf, Leaf+ Low-Degree=Flip+ Leaf+ Low-Degree.

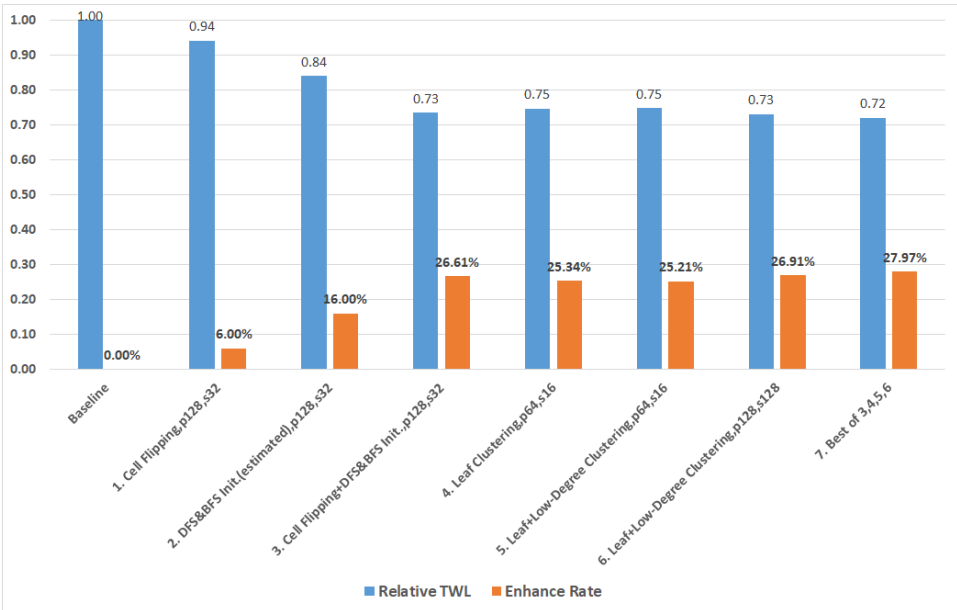


Figure 14 Overall enhancement rate comparison between Cell Flipping, DFS&BFS Initialization, Leaf, and Leaf+ Low-Degree in 502 units test

In Table 3, Baseline means the result by human expert and Cell Flipping means that the Cell Flipping algorithm is applied to 502 Units, and Leaf means that nodes that meet the condition of node degree 2 or less are clustered from leaf nodes in the graph, and then tested using the netlist. Leaf+Low-Degree means that in addition to the leaf clustering condition, nodes with a node degree of 2 are found and clustered inside of the graph, and then tested using the netlist. From the table, we can see that even with small populations and small selections, the Clustering algorithm can yield better results in some units. In Fig. 14, we can see the performance of each algorithm and the performance of algorithm combinations. In the case of DFS&BFS Initialization test, 124 units are used and the performance of 502 units test are estimated from the result. And once again, we can see that the larger the population and selection, the better the result. Extracting and merging the best results from each test case, we achieved 27.97% improvement. Due to the nature of genetic algorithms using randomness, the more tests are performed, the better the probability of getting better results, even under the same test conditions.

6.6 Parallel Processing

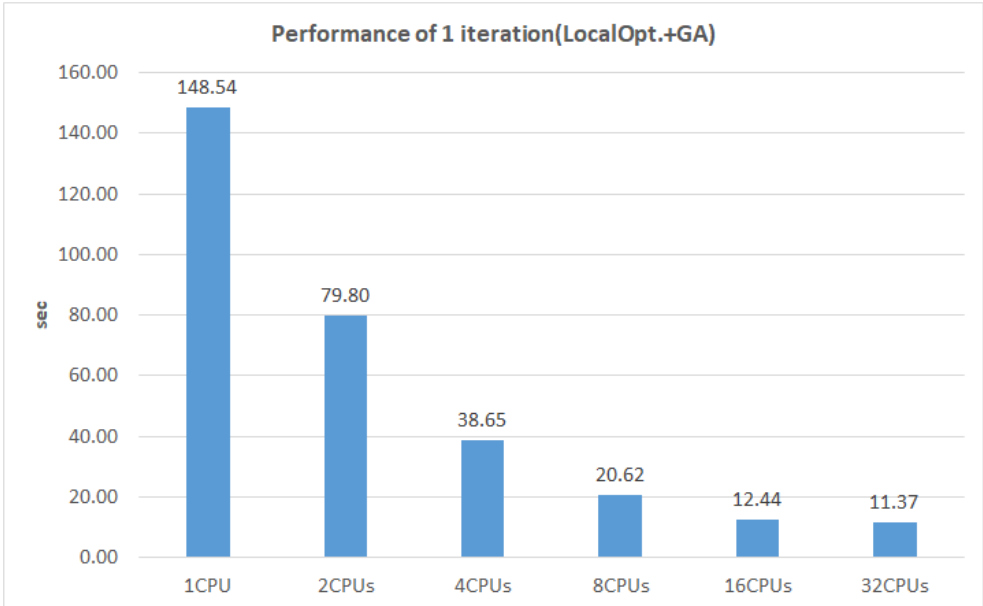


Figure 15 Parallelized performance according to the number of CPUs(IntelXeon E5-2620 v4@2.10GHz, 16 core 32 threads). Test condition: Cell length=29, Population=128, number of selection=32, iteration=2,000

We parallelized the operations from Selection to *Local Optimization* in Fig. 15 using multiple CPU cores and achieved 13.06x faster performance compared to a single-core execution environment in the 200 inverter chain test.

Chapter 7 Conclusion

We first addressed the 1d placement problem in depth for the full custom design flow. To minimize the total wire length of the units consisting of 3 to 98 standard cells, we devised a novel *Local Optimization* algorithm, *Greedy-K*, and it shows distinguished performance. And we identified that *Cell Flipping* and *DFS&BFS Initialization* algorithms significantly contribute to minimize the TWL. Using these techniques, we achieved a 27.97% improvement in TWL compared to human experts. We also found that parallel processing technique is essential for fast runtime.

Bibliography

[1] Charles M Fiduccia and Robert M Mattheyses. 1988. A linear-time heuristic for improving network partitions. In Papers on Twenty-five years of electronic design automation. 241–247.

[2] Keld Helsgaun. 2017. An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems. Roskilde: Roskilde University (2017), 24–50.

[3] Lalin L Laudis, Shilpa Shyam, V Suresh, and Ajay Kumar. 2018. A study: Various NP-hard problems in VLSI and the need for biologically inspired heuristics. In Recent findings in intelligent computing techniques. Springer, 193–204.

[4] Shen Lin and Brian W Kernighan. 1973. An effective heuristic algorithm for the traveling-salesman problem. Operations research 21, 2 (1973), 498–516.

[5] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. 2021. Reinforcement learning for combinatorial optimization: A survey. Computers & Operations Research 134 (2021), 105400.

[6] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Jiang, Ebrahim Songhori, ShenWang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Sungmin Bae, et al. 2020. Chip placement with deep reinforcement learning. arXiv preprint arXiv:2004.10746 (2020).

[7] Soheil Nazar Shahsavani, Alireza Shafaei, and Massoud Pedram. 2018. A placement algorithm for superconducting logic circuits based on cell grouping and super-cell placement. In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 1465–1468.

[8] Xiaoping Tang, Ruiqi Tian, and Martin DF Wong. 2005. Optimal redistribution of white space for wire length minimization. In Proceedings of the 2005 Asia and South Pacific Design Automation Conference. 412–417.

[9] Junchi Yan, Xianglong Lyu, Ruoyu Cheng, and Yibo Lin. 2022. Towards Machine Learning for Placement and Routing in Chip Design: a Methodological Overview. arXiv preprint

arXiv:2202.13564 (2022).

Abstract

면적 감소는 웨이퍼당 Netdie 증가로 수익성을 높이기 때문에 반도체 설계에서 가장 중요한 목표 중 하나이다. ASIC 설계를 위한 면적 최적화를 지원하는 다수의 상업용 툴이 존재하지만, 메모리 설계 분야에서는 Full Custom Design Flow를 이용해서 chip의 면적을 줄이고 있다. 이는 설계 관련 데이터에 대한 공유를 엄격히 제한하는 메모리 설계 회사의 보안 규정으로 인해, 상용 툴 제작 업체가 메모리 설계 분야에 관련된 최적화 툴을 개발하지 못했음에 기인한다. 특히, 메모리의 Peripheral 영역을 설계하기 위해서 표준 셀을 1차원(1D)으로 배치하는 절차가 존재하는데, 이 또한 지원하는 툴이 없는 실정이다. 위의 내용에서 영감을 받아 표준 셀의 1차원 배치를 위한 레이아웃에서 와이어 길이를 최소화하는 새로운 프레임워크를 제안한다.

표준 셀을 1차원으로 배치하기 위한 프레임워크는 다음과 같은 알고리즘으로 구성되는데, 여러 표준 셀로 구성된 1D 단위 블록의 전체 와이어 길이를 최소화하기 위해 표준 셀을 효율적으로 배치하는 휴리스틱 알고리즘, *Clustering* 알고리즘 및 클럭 제거-재구성 알고리즘이 프레임워크의 그 핵심이다. 이 세 가지 알고리즘을 적용함으로써, 전문가들이 설계한, 3~98개의 표준 셀로 구성된 502개 유닛 블록의 총 와이어 길이를 27.97% 개선하는 성과를 달성했다.