Population Studies for the Gate Matrix Layout Problem

Alexandre Mendes, Paulo França, Pablo Moscato and Vinícius Garcia

Departamento de Engenharia de Sistemas
Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas
C.P. 6101 - 13083-970 - Campinas - SP - Brazil
{smendes, franca, moscato, jacques}@densis.fee.unicamp.br

Abstract. This paper deals with a Very Large Scale Integrated (VLSI) design problem that belongs to the NP-hard class. The Gate Matrix Layout problem has strong applications on the chip-manufacturing industry. A Memetic Algorithm is employed to solve a set of benchmark instances, present in previous works in the literature. Beyond the results found for these instances, another goal of this paper is to study how the performance of the algorithm is affected by the use of multiple populations, together with different individual-migration policies. This comparison has shown to be fruitful, sometimes producing a strong performance improvement of the multiple populations approaches over the single population ones.

1 Introduction

The use of multiple populations in Evolutionary Algorithms (EAs) gained increased momentum when computer networks, multi-processors computers and distributed processing systems (such as workstations clusters) became widespread available. Regarding the software issue, the introduction of PVM, and later MPI, as well as web-enabled object-oriented languages like Java also had their role. As most EAs are inherently parallel methods, the distribution of the tasks is relatively easy for most applications. The workload can be distributed at an individual or a population level; the final choice depends on how complex are the computations involved. In this work we do not use parallel computers, or networks of workstations. The program runs in a sequential way on a single processor, but populations evolve separately, simulating the behavior of a parallel environment. With several populations evolving in parallel, larger portions of the search space can be sampled, and any important information found can be communicated among them through migration of individuals. This makes the parallel search much more powerful than when a single population is employed.

2 A VLSI optimization problem: Gate Matrix Layout

The Gate Matrix Layout problem is a NP-hard problem [3] that arises in the context of physical layout of Very Large Scale Integration (VLSI). It can be stated as: suppose that there are *g* gates and *n* nets on a gate matrix layout circuit. Gates can be described as vertical wires holding transistors at specific positions with nets interconnecting all the distinct gates that share transistors at the same position. An instance can be represented as a 0-1 matrix, with *g* columns and *n* rows. A number one in the position (*i*, *j*) means a transistor must be implemented at gate *i* and net *j*. Moreover, all transistors in the same net must be interconnected. This superposition of interconnections defines the number of tracks needed to build the circuit. The objective is to find a permutation of the *g* columns so that the superposition of interconnections is minimal, thus minimizing the number of tracks. The figure below shows a possible solution for a given instance, and how to go from the 0/1 representation to the circuit itself.

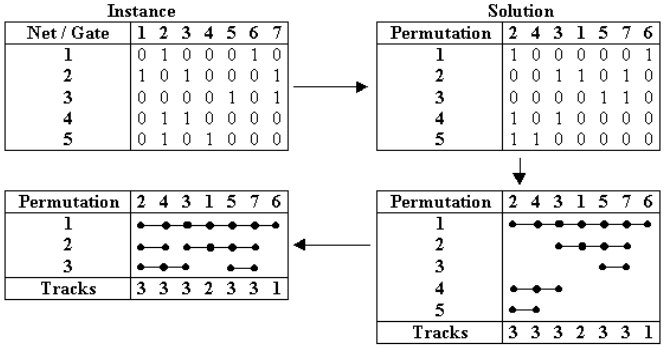

Fig. 1. The translation from a given instance's solution into the real circuit.

In the example, the permutation of the columns was <2-4-3-1-5-7-6>. After the interconnection of all transistors, represented by the horizontal lines, we calculate the number of tracks needed to build each gate. This number is the sum of positions used in each column and the number of tracks needed to build the circuit is its maximum. More detailed information on this problem can be found in [4].

3 Memetic Algorithms

Since the publication of John Holland's book, "*Adaptation in Natural and Artificial Systems*", the field of *Genetic Algorithms*, and the larger field of *Evolutionary Computation*, was clearly established as new research areas. However, other pioneer works would also be cited, but since Holland's work they became increasingly conspicuous in many engineering fields and in Artificial Intelligence problems. In the mid 80's, a new class of *knowledge-augmented GAs*, also called *hybrid GAs*, started

to appear in the computer science literature. The main idea supporting these methods is that of making use of other forms of "knowledge", i.e. other solution methods already available for the problem at hand. As a consequence, the resulting algorithms had little resemblance with biological evolution analogies. Recognizing important differences and similarities with other population-based approaches, some of them were categorized as *Memetic Algorithms* (MAs) in 1989 [7][9].

3.1 Population structure

It is illustrative to show how some MAs resemble more the cooperative problem solving techniques that can be found in some organizations. For instance, in our approach we use a *hierarchically structured population* based on a complete ternary tree. In contrast with a non-structured population, the complete ternary tree can also be understood as *a set of overlapping sub-populations* (that we will call clusters).

In Figure 2, we can see that each cluster consists of one single *leader* and three *supporter* individuals. Any leader individual in an intermediate layer has both leader and supporter roles. The leader individual always contains the best solution – considering the number of tracks it requires – of all individuals in the cluster. The number of individuals in the population is equal to the number of nodes in the ternary tree, *i.e.*, we need 13 individuals to make a ternary tree with 3 levels and 40 individuals to have 4 levels.

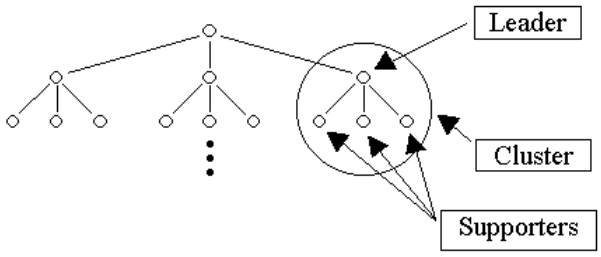

Fig. 2. Diagram of the population structure.

Previous tests comparing the tree-based population with the non-structured approach are present in 1. They show that the ternary-tree approach leads to better results, and with the use of a smaller number of individuals. In this work, we could not address this issue again due to space limitations.

3.2 Representation and crossover

The representation chosen for the VLSI problem is quite intuitive. A solution is represented as a "chromosome". The alleles assume different integer values in the [1, *n*] interval, where *n* is the number of columns of the associated matrix. The crossover tested is a variant of the well-known *Order Crossover* (OX), called *Block Order Crossover* (BOX). After choosing two parents, several fragments of the

chromosome from one of them are randomly selected and copied into the offspring. In the second phase, the offspring's empty positions are sequentially filled according to the chromosome of the other parent. The procedure tends to perpetuate the *relative order* of the columns, although some alterations might appear.

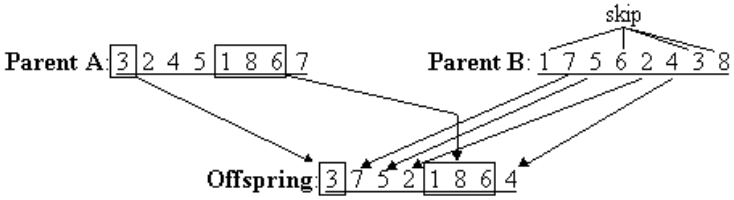

Fig. 3. Block Order Crossover (BOX) example.

In Figure 3, *Parent A* contributes with two pieces of its chromosome to the offspring. These parts are thus copied to the same position they occupy in the parent. The blank spaces are then filled with the information of *Parent B*, going from left to right. Numbers in *Parent B* already present in the offspring are skipped; being copied only the new ones. The contribution percentage of each parent is set to be 50%. This means that the offspring will be created from information inherited in equal proportion from both parents.

The number of new individuals created every generation is two times the number of individuals present in the population. This crossover rate, apparently high, is due to the offspring acceptance policy. The acceptance rule makes several new individuals be discarded. Thus after several tests, with values from 0.5 to 2.5 we decided to use 2.0. The insertion of new solutions in the population will be later discussed (see section 3.6).

3.3 Mutation

A traditional mutation strategy based on swapping of columns was implemented. Two positions are selected uniformly at random and their values are swapped. This mutation procedure is applied to 10% of all new individuals every generation.

We also implemented a *heavy mutation* procedure. It executes the job swap move *10.n* times in each individual – where *n* is the number of gates - except the best one. This procedure is executed every time the population diversity is considered to be low, *i.e.*, it has converged to individuals that are too similar (see section 3.6).

3.4 Local Search

Local search algorithms for combinatorial optimization problems generally rely on a neighborhood definition that establishes a relationship between solutions in the configuration space. In this work, two neighborhood definitions were chosen. The first one was the *all-pairs*. It consists of swapping pairs of columns from a given

solution. A *hill-climbing* algorithm can be defined by reference to this neighborhood; *i.e.*, starting with an initial permutation of all columns, every time a proposed swap reduces the number of tracks utilized, it is confirmed and another cycle of swaps takes place, until no further improvement is achieved. As the complexity to evaluate each swap is considerably high, we used a reduced neighborhood, where all columns are tested for a swap, but only with the closer ones. Following this, we try swapping all columns only with their 10 nearest neighbors. This number is not so critical, but we noticed a strong degradation in performance when values around 5 or lower were utilized.

The second neighborhood implemented was the *insertion* one. It consists of removing a column from one position and inserting it in another place. The hill-climbing iterative procedure is the same regardless the neighborhood definition. In this case, we also utilized a reduced neighborhood. Each column was tested for insertion only in the 10 nearest positions.

Given the large size of the all-pairs and insertion neighborhoods, and the computational complexity required to calculate the objective function for each solution, we found it convenient to apply the local search only into the best individual, located at the top node of the ternary tree, just before migration occurs.

3.5 Selection for recombination

The recombination of solutions in the hierarchically structured population can only be made between a leader and one of its supporters within the same cluster. The recombination procedure selects any leader uniformly at random and then it chooses – also uniformly at random – one of the three supporters.

3.6 Offspring insertion into the population

After the recombination, mutation and local search phases have finished, the acceptance of the new offspring will follow two rules:

- The offspring is inserted into the population *replacing the supporter* that took part in the recombination *that generated it*.
- The replacement occurs *only if* the fitness of the new individual *is better than* the supporter.

If during the recombination phase no individual was accepted for insertion, we conclude that the population has converged and *apply the heavy mutation procedure*. Finally, after the recombination phase, the population is restructured. The hierarchy states that the fitness of the leader of a cluster must be lower than the fitness of the leader of the cluster just above it. The adjustment is done comparing the supporters of each cluster with the leader. If any supporter turns out to be better than its respective leader, they swap their places. Considering the problem addressed in this work, the higher is the position that an individual occupies in the tree, the fewer is the number of tracks it utilizes.

4 Migration policies

For the study with multiple populations, we had to define how individuals migrate from one population to another. There are three population migration policies:

- **0-Migrate**: No migration is used and all populations evolve in parallel without any kind of communication or solutions exchange.
- **1-Migrate**: Populations are arranged in a ring structure. Migration occurs in all populations and the best individual of each one migrates to the population right next to it, replacing a randomly chosen individual – except the best one. Every population receives only one new individual.
- **2-Migrate**: Populations are also arranged in a ring structure. Migration also occurs in all populations, but the best individual of each one migrates to both populations connected to it, replacing randomly chosen individuals – except the best ones. Every population receives two new individuals.

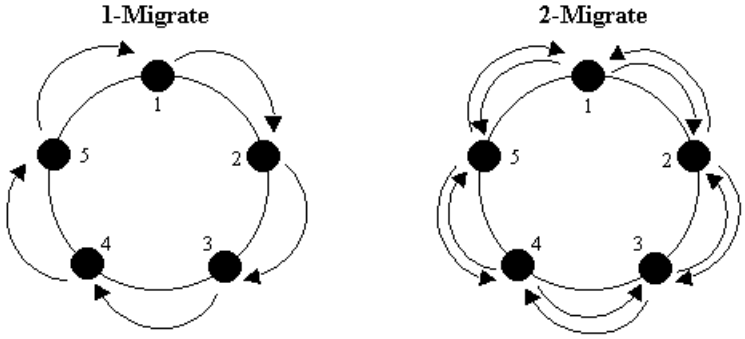

Fig. 4. Diagrams of the two migration policies.

5 Computational tests

The population tests were executed in two different ways. The first one was to test the influence of the number of populations on the performance. For this evaluation the number of populations varied from one up to five. The second test evaluated the influence of migration on the algorithm's performance. For each number of populations we tested the three migration policies, totaling 15 configurations. The tests are divided into instances, *i.e.*, for each instance we tested the whole set of configurations, ten times each one. Five instances were tested. The stop criterion was a time limit, fixed as follows: 30 seconds for W2, V4470 and X0; 90 seconds for W3 and 10 min for W4. The difference of maximum CPU times is due to the dimension of the instances and takes into account the average time to find high quality solutions. If larger CPU times were utilized, most configurations would return excellent results, weakening any possible comparison among them.

In Table 1 we show some information on the instances we utilized in this work. We have one small, three medium and a large instance. In the literature it is difficult to find hard instances. In ref. [4], we found the most extensive computational tests, with 25 instances in total. However, most of them were too small and easy to solve with our algorithm. Considering the instances' sizes, only V4470, X0, W2, W3 and W4 had more than 30 gates and for this reason we centered our studies on them.

Table 1. Information on the instances.

Instance	Gates	Nets	Best known solution
W2	33	48	14
V4470	47	37	9
X0	48	40	11
W3	70	84	18
W4	141	202	27

Next we show the results of the MA implemented. Four numbers are utilized do describe the results for each configuration (see Figure 5). In clockwise order we have: In boldface, the best solution found for the number of tracks for that instance. Next in the sequence we display the number of times this solution was found in ten tries. Below it, there is the worst value found for the configuration, and finally, in the lower-left part of the cell, is the average value found for the number of tracks.

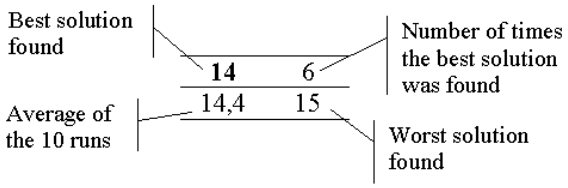

Fig. 5. Data fields for each configuration.

All tests were executed in a PENTIUM 366 MHz Celeron computer, using Sun JDK 2.0 Java language under Windows environment.

Table 2. Results for the W2 instance.

W2	Number of populations									
	1		2		3		4		5	
0-Migrate	**14**	9	**14**	10	**14**	8	**14**	9	**14**	5
	14.1	15	14.0	14	14.2	15	14.1	15	14.5	15
1-Migrate			**14**	10	**14**	9	**14**	8	**14**	5
			14.0	14	14.1	15	14.2	15	14.5	15
2-Migrate			**14**	9	**14**	5	**14**	6	**14**	4
			14.1	15	14.5	15	14.4	15	14.6	15

Table 3. Results for the V4470 instance.

V4470	**Number of populations**									
	1		**2**		**3**		**4**		**5**	
0-Migrate	**9**	2	**9**	1	**9**	2	**10**	10	**10**	9
	10.1	11	10.0	10	10.0	11	10.0	10	10.1	11
1-Migrate			**9**	2	**9**	1	**9**	2	**10**	8
			10.1	11	10.0	11	9.9	11	10.2	11
2-Migrate			**9**	2	**9**	1	**9**	1	**10**	6
			9.9	11	10.2	11	10.1	11	10.4	11

Table 4. Results for the X0 instance.

X0	**Number of populations**									
	1		**2**		**3**		**4**		**5**	
0-Migrate	**11**	7	**11**	6	**11**	7	**11**	7	**11**	3
	11.4	13	11.4	12	11.3	12	11.5	13	11.7	12
1-Migrate			**11**	6	**11**	6	**11**	6	**11**	3
			11.5	13	11.6	13	11.4	12	11.9	13
2-Migrate			**11**	7	**11**	4	**11**	3	**11**	5
			11.5	13	12.0	13	11.9	13	11.8	14

Table 5. Results for the W3 instance.

W3	**Number of populations**									
	1		**2**		**3**		**4**		**5**	
0-Migrate	**18**	1	**18**	2	**18**	1	**19**	1	**19**	2
	21.1	26	20.1	23	20.1	22	20.3	22	20.8	22
1-Migrate			**18**	3	**18**	1	**18**	2	**18**	2
			20.0	23	20.1	23	20.2	22	20.0	22
2-Migrate			**18**	1	**18**	2	**18**	1	**18**	1
			20.2	23	21.1	25	20.3	23	21.3	23

Table 6. Results for the W4 instance.

W4	**Number of populations**									
	1		**2**		**3**		**4**		**5**	
0-Migrate	**29**	3	**29**	2	**29**	2	**30**	4	**32**	1
	31.5	36	31.1	33	31.4	35	32.1	36	33.7	35
1-Migrate			**28**	2	**28**	1	**28**	2	**29**	3
			31.4	34	31.2	35	30.6	35	31.9	36
2-Migrate			**29**	2	**29**	3	**29**	1	**31**	1
			32.5	36	31.7	35	33.4	35	35.1	38

First, we should explain two aspects of randomized search algorithms: exploitation and exploration. Exploitation is the property of the algorithm to thoroughly explore a specific region of the search space, looking for any small improvement in the current best available solution(s). Exploration is the property to explore wide portions of the search space, looking for promising regions.

With no migration, we observed more instability in the answers, expressed by worst solutions and averages found for the instances W3 and W4. On the other hand, the 1-Migrate appeared to better balance exploitation and exploration, with good average and worst-solution values. The 2-Migrate policy did not perform so well, with a clear degradation of these two parameters. A too strong exploitation, in detriment of the exploration shall have caused this. Thus we concluded that migration should be set at medium levels, represented by the 1-Migrate.

The second aspect to be analyzed is the number of populations. Although it is not clear what configuration was the best, the use of only one is surely not the best choice since several multi-population configurations returned better values.

All the values previously found in the literature, presented in Table 1, were reached by the MA, except the W4. An increase in the CPU time to 60 minutes to check if the 27 tracks value did also not work out. That time limit was determined after we verified that the best results for W4, presented in [5], took several hours to be achieved even using an equipment which performance was comparable to ours.

As even with a long CPU time the algorithm did not succeed, we decided to enlarge the local search neighborhood, increasing from the number of positions to be tested from 10 to 20 (see section 3.4). With this change the algorithm finally succeeded to find the best solution three times in ten.

Table 7. Results for instance W4 with a 60-minutes CPU time limit and the augmented local search neighborhood.

Test #	Best solution found	CPU time required to reach the solution
1	27	2141.0
2	28	1791.9
3	29	855.7
4	29	532.4
5	27	1788.7
6	29	560.8
7	28	1791.0
8	31	2908.1
9	27	3002.5
10	28	600.9

6 Conclusions

This work presented a study on multiple-population approaches to solve the gate matrix layout problem. We used a Memetic Algorithm as the search engine. The results were very encouraging and the best multi-population configuration found was four populations evolving in parallel and exchanging individuals at a medium rate. The five instances utilized were taken from real-world VLSI circuit layout problems and the solutions rivaled with those previously found in the literature. Another strong point is that the method utilized is included in a framework for gen-

eral optimization called NP-Opt [6]. That means a general purpose MA was successful in solving a very complex optimization problem, competing head-to-head with specific methods especially tailored for this problem. Future works should include the use of parallel techniques to distribute the populations and/or individuals through a computer network and the extension of this study to other NP problems to verify if the results hold as well.

Acknowledgements

This work was supported by "Fundação de Amparo à Pesquisa do Estado de São Paulo" (FAPESP – Brazil) and "Conselho Nacional de Desenvolvimento Científico e Tecnológico" (CNPq – Brazil). The authors also thank Alexandre Linhares, for his relevant remarks and comments, and for providing us with the VLSI instances.

References

1. França, P. M., Mendes, A. S. and Moscato, P.: A memetic algorithm for the total tardiness Single Machine Scheduling problem. European Journal of Operational Research, v. 132, n. 1 (2001) 224-242
2. Hu, Y. H. and Chen, S. J.: GM_Plan: A gate matrix layout algorithm based on artificial intelligence planning techniques. IEEE Transactions on Computer-Aided Design, v. 9 (1990) 836-845
3. Lengauer, T.: Combinatorial algorithms for integrated circuit layout. John Wiley & Sons, New York (1990)
4. Linhares, A.: Synthesizing a Predatory Search Strategy for VLSI Layouts, IEEE Transactions on Evolutionary Computation, v. 3, n. 2 (1999) 147-152
5. Linhares, A., Yanasse, H. and Torreão, J.: Linear Gate Assignment: a Fast Statistical Mechanics Approach. IEEE Transactions on Computer-Aided Design on Integrated Circuits and Systems, v. 18, n. 12 (1999) 1750-1758
6. Mendes, A. S., França, P. M. and Moscato, P.: NP-Opt: An Optimization Framework for NP Problems. Proceedings of POM2001 - International Conference of the Production and Operations Management Society (2001) 82-89
7. Moscato, P.: On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms. Caltech Concurrent Computation Program, C3P Report 826. (1989)
8. Nakatani, K., Fujii, T., Kikuno, T. and Yoshida, N.: A heuristic algorithm for gate matrix layout. Proceedings of International Conference of Computer-Aided Design (1986) 324-327
9. Moscato, P. and Norman, M. G.: A 'Memetic' Approach for the Traveling Salesman Problem. Implementation of a Computational Ecology for Combinatorial Optimization on Message-Passing Systems, Parallel Computing and Transputer Applications, edited by M. Valero, E. Onate, M. Jane, J.L. Larriba and B. Suarez, Ed. IOS Press, Amsterdam (1992) 187-194