18. VLSI design: gate matrix layout problem

Pablo Moscato, Alexandre Mendes and Alexandre Linhares

18.1 Introduction

With applications ranging from fields as distinct as fuzzy modeling (Xiong 2001), autonomous robot behavior (Luk et al. 2001), learning with backpropagation (Foo et al. 1999), and multicriteria optimization (Viennet et al. 1996), evolutionary methods have become an indispensable tool for systems scientists. Although already studied in the past, an interesting emerging issue is the use of multiple populations, which is gaining increased momentum from the conjunction of two technologies. On the hardware side, computer networks, multi-processor computers and distributed processing systems (such as workstations clusters) are increasingly becoming widespread. Regarding the software issues, the introduction of Parallel Virtual Machine\(^1\) (PVM), and later the Message Passing Interface Standard\(^2\) (MPI), as well as web-enabled, object-oriented languages (such as Java\(^3\)) have also had their role. Most Evolutionary Algorithms (EAs) are methods that are easy to parallelize as well as naturally suitable for heterogeneous systems. For most EAs 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 depending on the complexity of the computations involved.

The application presented in this chapter is motivated by the availability of these computer environments. However, here we do not report results on the use of parallel computers, or networks of workstations. The proposed memetic algorithm (MA) runs in a sequential way on a single processor, but a set of populations evolve separately, an approach that can be easily mapped to a parallel environment.

Species evolve naturally grouped in sub-populations, with boundaries defined by some specific features like distance or geographical barriers. The role of closed (or nearly closed) populations in biological evolution is extremely important. Consider for instance the Galapagos Islands, an example of notorious inspirational role for Darwin’s ideas when aboard the HMS Beagle (Darwin 1993). A set of islands separated by several kilometers of water can be colonized by a single species of birds. In the beginning of such colonization, all animals will share the same characteristics (and genetic pool), but as evolution takes place, the groups concentrated in each island will start to differentiate by adapting themselves to the particular characteristics present in each island (Weiner 1995). This independent adaptation

\(^{1}\) http://www.csm.ornl.gov/pvm/pvm_home.html
\(^{2}\) http://www-unix.mcs.anl.gov/mpi/
\(^{3}\) JAVA is a general programming language developed by SUN Microsystems.
may eventually lead to having different species after a sufficient number of gener-
ations, given very little or no migration at all exists between the islands. Even if
the islands share the same characteristics, different species might arise, due to the
relative isolation and the genetic drift phenomenon (Weiner 1995).

The genetic drift concept states that if two identical populations are separated
and submitted to equal conditions, due to the random nature of the processes in-
volved in evolution, they may still follow different evolutionary paths and become
different species after a large number of generations. Analogously, it is quite
common in EAs to find that, due to the random nature of the approach, if the same
algorithm is run twice it may generate different final solutions. Usually viewed as
a setback, this characteristic can actually be very useful when multiple populations
are used. With several populations evolving in parallel, larger portions of the
search space can be sampled, and any particularly important new features found
could be spread among them through migration of individuals. This mechanism
makes the parallel search potentially more powerful than single population ap-
proaches (Cantú-Paz 1997, 1999).

18.2 The gate matrix layout problem

The Gate Matrix Layout problem (GMLP) belongs to the NP-hard class
(Lengauer 1990; Linhares 1999; Nakatani et al. 1986) and arises in the context of
the physical layout of Very Large Scale Integration (VLSI) systems. Formally we
can state the problem as:

**Instance:** A matrix $M$ having Boolean coefficients.

**Parameter:** $p > 0$, integer.

**Question:** Is there a permutation of the columns of $M$ so that if in each row we
change to ‘*’ every 0 lying between the row’s leftmost and rightmost 1, then no
column contains more than $p$ 1’s and *’s?

This problem has appeared in the literature in a number of combinatorially
equivalent forms. The form given above was defined in (Lopez and Law 1980). It
has been known as ‘Weinberger arrays’ or ‘PLAs with multiple folding’. The
problem is solvable in $O(n)$ for fixed $p$ (Bodlaender 1993, 1996; Fellows and
Langston 1989).

We can now state the problem in the terms of the familiar VLSI setting. Sup-
pose that there are $g$ gates (they can be described as vertical wires holding transis-
tors at specific positions) and $n$ nets (horizontal wires interconnecting all the dis-
tinct gates that share transistors at the same position). As described above, an
instance of the GMLP can be represented as 0-1 matrix, with $g$ columns and $n$
rows. An entry ‘1’ in the position $(i, j)$ means a transistor must be implemented at
gate $i$ and net $j$; 0 means that no such connection is to be made. The VLSI archi-
tecture requires that all transistors in the same net must be interconnected. Given a
specific gate sequence, whenever two nets overlap, their implementation needs
two separated physical tracks. This superposition of interconnections defines the number of tracks needed to build the circuit.

Both types of formulation are equivalent. The objective in the GMLP is to find the minimal \( p \). In other words, is to find a permutation of the \( g \) columns so that the superposition of interconnections is minimal, thus minimizing the number of tracks and the overall circuit area. The Fig. 18.1 shows a possible solution for an instance composed of seven gates and five nets, and how to go from the binary matrix representation to the circuit itself.

The Fig. 18.1 represents a circuit with seven gates and five nets. Usually, real circuit matrices are very sparse, like the one in the example. Based on the instance, we created a feasible solution, represented by the columns permutation (2-4-3-1-5-7-6). The translation from the 0-1 representation into the circuit diagram is very simple. All the values ‘1’ in the same net must be interconnected (represented by the horizontal lines). In the lower part of this diagram there is the sum of the transistors in each gate. Finally, the real circuit with only three tracks remaining, after the transistor-grouping routine was applied, is shown in the lower-left part of the Fig. 18.1.

For more information on this problem, including other industrial settings in which it arises, please refer to (Wong et al. 1988; Yanasse 1997; Linhares and Yanasse 2002). The reader should be aware that this is not just an ordinary NP-hard problem: it was in fact one of the first problems identified as being fixed-parameter tractable, and this result eventually led to the creation of a new, large class of problems under the label FPT, for fixed parameter tractability (Downey and Fellows 1995; Fellows and Langston 1987). In the next section we introduce a new MA for the GMLP.
18.3 The memetic algorithm

Since the publication of John Holland’s book, ‘Adaptation in Natural and Artificial Systems’ (Holland 1975), the field of Genetic Algorithms (GAs), and the broader field of Evolutionary Computation (EC), were clearly established as new research areas. However, other pioneering works could also be cited, as they became increasingly conspicuous in many engineering fields and in industrial problems. In the mid 80’s, a new class of knowledge-augmented GAs, also called hybrid GAs, began to appear in the computer science and engineering 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 in 1989 (Moscato 1989; Moscato and Norman 1992), a denomination inspired from the term meme introduced by Dawkins (Dawkins 1976). The field of cultural evolution was suggested as being more relevant, as a working metaphor, to understand the performance and find inspiration sources to improve these new methods. Next we will describe the main aspects of the MA implemented.

18.3.1 Population structure

The use of structured populations in evolutionary algorithms is still quite rare. Nevertheless, previous tests (França et al. 2001; Mendes et al. 2001) in different types of NP optimization problems have shown that this feature can generate a big leap in the method’s performance. It was verified that the number of individuals utilized by the method is dramatically reduced when population structures such as binary or ternary trees are employed. The consequence is a reduction in the computational effort and a considerable performance improvement, when compared to ordinary non-structured population approaches (França et al. 2001; Mendes 1999).

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 the approach being described, 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 (which we will refer to as clusters). The choice of the ternary tree structure was based mainly on empirical aspects. The first is motivated by the fact that any hierarchical tree behaves like a set of overlapping clusters, as said before. Therefore, the dynamics is similar to several populations evolving in parallel – each cluster acts as an independent population – and exchanging individuals at a given rate. This exchange of individuals comes as a consequence of the tree-restructuring phase, carried out to maintain a specific hierarchical consistence. Now, consider the use of trees with other degrees of complexity. A binary tree-based population, for instance, would be formed by 3-individual clusters only, with only two recombinations possibilities. This would degrade the ‘multiple population’ character of
the tree structure. Trees with a greater order – quaternary or more – increase the multiple population character, but initial tests indicated that the performance does not improve at all and moreover, the number of individuals rapidly jumps to prohibiting levels in terms of computational effort requirements. The best trade-off points to the selected ternary tree structure.

![Diagram of the ternary tree hierarchical structure utilized by the MA populations](image)

**Fig. 18.2.** Diagram of the ternary tree hierarchical structure utilized by the MA populations

In Fig. 18.2, 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 hierarchy states that the leader individual must always contain the best solution – considering the number of tracks – present 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 three levels and 40 individuals to have four levels. For this experiment, we fixed the population size to be 13. This value might seem too low at a first glance, but after several tests with 40 and 121 individuals, we concluded that 13 individuals are sufficient to make the algorithm keep its convergence speed under control. The use of 40 or more individuals does not deteriorate the algorithm’s behavior, but the computational effort increases considerably, as well as the CPU time. Moreover, the MA with the 13-individual configuration had a very competitive performance with other types of problems (França et al. 2001; Mendes et al. 1999).

### 18.3.2 Representation and crossover

The success of a MA relies very much on the representation chosen to encode for feasible solutions, and to follow the terminology of genetic algorithms we will call them “chromosomes”. It is preferable to use representations that are compact, complete and stable. A compact representation requires very few variables to represent a solution in a unique way. A complete representation must be able to represent all the possible solutions of the problem, including, of course, the optimal one. A stable representation is one in which small changes in the chromosome structure would result in solutions with small modifications in the individual’s fitness. If small disturbances generate large fitness modifications, the MA might have trouble to evolve the population because important information learned during the evolutionary process is continually lost. The representation chosen for the
problem is quite intuitive. A solution is represented as a chromosome in which the genes\(^4\) assume different integer values in the \([1, g]\) interval, where \(g\) is the number of columns of the associated binary matrix. These values will define the sequence (permutation) of the gates.

The recombination algorithm used to address the GMLP is a variant of the well-known Order Crossover (OX) (Goldberg 1989) called Block Order Crossover (BOX). The OX is very simple and easy to implement, but lacks an essential feature for the GMLP. It begins by copying a piece of the chromosome from one of the parents into the offspring. This can already be a major limitation if the good features of the individuals are not grouped together, but separated in many blocks. In this case, a better-tailored crossover should be able to copy several parts of the chromosome into the offspring. This is the motivation for the BOX, where instead of just one piece, several pieces of the parent are copied. The BOX resembles the second variant of the OX crossover presented in (Syswerda 1991).

The rest of the BOX crossover is the same of the OX: The loci\(^5\) in the offspring that were not filled with the information of the first parent are sequentially filled out with the alleles inherited from the second parent. Both procedures – OX and BOX – tend to perpetuate the relative order of the gates, but the BOX can better communicate separated blocks to the offspring. Nevertheless, given the stochastic characteristic of the crossover process, undesirable alterations might – and will – appear during the evolutionary process, as it happens in nature.

\[\text{Parent A: } [3, 2, 4, 5, 1, 8, 6, 7] \quad \text{Parent B: } [1, 7, 5, 6, 2, 4, 3, 8]\]

\[\text{Offspring: } [3, 5, 2, 1, 8, 6, 4]\]

**Fig. 18.3.** Block Order Crossover (BOX) example

In Fig. 18.3, **Parent A** contributes with two pieces of its chromosome to the offspring – one with a single allele and the other with three alleles. These parts are copied in the same positions that they occupy in the parent. The blank spaces are then filled with the information from **Parent B**, going from left to right. The values in **Parent B** already present in the offspring are skipped; being copied only the new ones. In the structured population approach, **Parent A** is always a leader and **Parent B** a supporter belonging to the same cluster.

\(^4\) Chromosomes are composed of ordered sequences of genes.

\(^5\) Loci (plural of locus) is the name given to the positions of the chromosome where the alleles are placed.
The contribution from each parent to the offspring is another important issue. In nature, parents contribute in different proportions to the offspring’s genetic material, given the randomized characteristics of the genetic recombination. It is somewhat common to see GA/MA work where the genetic inheritance is proportional to the parent’s fitness. In other words, parents better adapted will influence more their offspring’s genetic conformation. In this implementation we decided that each parent contributes with a fixed 50% of its genetic material. The decision of not using the fitness was based on the little influence it had on the results. This has probably occurred due to the use of a very strong local search. The local search capacity to improve the individuals’ fitness has made any little modification in the crossover characteristics be a negligible factor in the algorithm’s performance. Nevertheless, we must emphasize that the crossover is a crucial step of the evolutionary process; a situation confirmed by the considerable difference of performance between the OX and BOX crossovers in our preliminary tests.

The number of descendants created every generation is another important issue and can strongly influence the quality of the algorithm. Generally the number of descendants has a close relation to the percentage of them that will be included in the population, replacing the old ones. If all new individuals are accepted, the crossover rate shall be set at low levels, otherwise good information already present in the gene pool might be lost. If the offspring’s insertion policy is more restrictive, more individuals should be created, since only a few of them will survive. As the policy adopted is indeed very restrictive, we assumed that the crossover rates should be higher than normal. In fact, after tests with values ranging from 0.5 up to 2.5 times the population size, the best value found for the crossover rate was 2.0. This number corroborates the relation described before, characterizing a high offspring mortality selection process.

18.3.3 Mutation

Mutation plays a critical role in evolutionary algorithms and in memetic algorithms design in particular (Merz and Freisleben 1999). It consists of a random change in a part, or parts, of the individual’s genetic code. This random nature makes the mutation be much more destructive than constructive. Indeed, the great majority of mutations are destructive, creating worse or non-viable individuals. Such individuals are usually eliminated through natural selection and the results of disastrous mutations are not perpetuated. But the strong point is that, when successful, mutation can create good, complex features, or even make the individual jump out of evolution traps by improving its fitness. When a population is very similar in terms of genetic characteristics, making any further improvement almost impossible, we can say it became stuck in an evolution trap. A good mutation movement can put an end to this situation by creating a new better individual that could be virtually unreachable through recombination alone.

6 The local search is described in Sect. 18.3.4.
7 The offspring’s insertion policy is described in Sect. 18.3.6.
Two mutation strategies are utilized in the MA. The first is a *light mutation*, which makes slight changes in the chromosome. It is based on the swapping of gates. Two positions of the chromosome are selected uniformly at random and the values of their alleles are swapped. This mutation procedure is applied (in average) on 10% of all new individuals every generation (see Fig. 18.4).

![Fig. 18.4. Swap mutation example. Two positions of the chromosome are selected uniformly at random and their alleles are swapped](image)

The second mutation strategy is called *heavy mutation*. Since the population continuously loses diversity during the evolutionary process, after a sufficient number of generations all individuals will share almost the same genetic code. When that happens, we say that the population has converged to a low-diversity situation. In such a situation, any further improvement is too costly. A heavy mutation procedure makes the population be filled with new individuals and a diversified genetic load, allowing the MA to continue its search. The procedure adopted executes a light mutation move exactly 10.g times in each individual, except the best one. The resulting population will thus be composed of several randomized, low-quality individuals, and a highly adapted one.

### 18.3.4 Local search

The local search is one of the main differences between this chapter’s MA and an ordinary GA. Its influence on the algorithm’s performance is overwhelming, being quite hard to think of a method that does not utilize a neighborhood-based search and can deal with problems as complex as the GMLP.

The design of local searches strategies is a rather complicated job. On one hand, we try to get as much search power as possible and on the other we must keep an eye on the increasing computational complexity. Next there is a step-by-step description of the local search construction, highlighting the mistakes and successes during the process.

Local search algorithms for combinatorial optimization problems generally rely on a neighborhood definition that establishes a relationship between solutions in

---

8 The definition of population convergence adopted this work is based on the individual’s replacement rate, rather than on chromosome similarities. Both criteria are related but the first is less restrictive. Please refer to Sect. 18.3.6.

9 The preservation of the best individual is a kind of *elitism*. It guarantees that the valuable genetic information present in its chromosome will not be lost during the evolutionary process.
the configuration space. Two neighborhood definitions were utilized in this specific local search. The first one was the all-pairs. It consists of swapping pairs of columns/gates 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 can be achieved. The second neighborhood implemented is named insertion. It consists of removing a column from one position and inserting it in another position (which could include any point between a pair of gates, or the left/right extremes of the permutation). The hill-climbing iterative procedure is the same regardless the neighborhood definition. Both neighborhoods have an $O(n^2)$ complexity. For small instances, with less than 30 gates, it is possible to evaluate every swap or insertion possibility, but since the goal was to develop a local search capable of running on a wide spectrum of instances, the use of neighborhood reductions became imperative.

Reduction techniques are designed to work with smaller neighborhood sizes without considerable loss of performance. Most reductions are based on general empirical conclusions and special characteristics of the problem. The first type utilizes less problem-specific information and its performance tends to be worse. Nevertheless, it is easier to implement them and can serve as an initial step to more complex reductions. That is the reason why we started with a simple reduction rule allowing the swap and insertion moves only between columns that are close enough to each other. The motivation for such approach is that the crossover and mutation operators are relatively successful in finding the region where each gate should be placed. No local search must be applied to make an allele swap its position with another allele on the opposite side of the chromosome. The evolutionary operators will sometime make such a change. But when a fine-tuning is required, crossover and mutation are useless. Local search must be employed in this case and the search can be concentrated only in the region being adjusted. In fact, the distance that characterizes such fine-tuning region is a critical parameter and should vary proportionally to the instance’s size. For instance, suppose there is an instance with 50 gates. A good choice observed in our preliminary tests was to fix the distance at $k = 10$, meaning that every column should be tested for swap/insertion with the 10 nearest columns to the right and the left, totaling 20 tests. Of course, larger values for $k$, like 20 or 30, should yield to better performances, but the computational complexity increases proportionally. This tradeoff analysis is a very stressing and time-consuming issue but also a necessary step. During this analysis, we experienced some problems with the largest instance, with 141 gates. The minimum $k$ values necessary to reach optimal solutions for it were still very high, making the search process too sluggish.

After the definition of the first reduction strategy, it became clear that a somewhat ‘smarter’ technique, utilizing specific GMLP information should be created. The resulting reduction is named Critical Column Based Local Search (CCBLS).

---

10 The $O(n^2)$ means that the computational complexity of visiting the entire neighborhood is proportional to the square of the instance’s size.
This new local search reduction scheme prohibits useless swaps and insertion tries by using information of the so-called critical columns. The critical columns define the maximum number of tracks required by the solution. In any given solution, every gate will have to implement a specific number of transistors. This number is given by the sum of the transistor-interconnecting horizontal lines that pass through the gate. After calculating this value for every column, we can define that column $i$ is critical if it satisfies the equation:

$$n_t(i) = \max_{\forall k \in G} \left[ n_t(k) \right]$$

(18.1)

where $n_t(\cdot)$ denotes the number of tracks in gate (\cdot) and $G$ is the set of gates.

In other words, the critical columns define the solution’s maximum number of tracks. The Fig. 18.5 shows an example of such columns in a given instance.

![Illustration of a critical column](image)

**Fig. 18.5.** Illustration of a critical column utilized in the CCBLS to reduce the overall computational effort.

In the Fig. 18.5, the critical column is given by gate number two, which requires six tracks to be implemented. The local search reduction policy works by prohibiting any swap or insertion that cannot affect the critical column(s). In the example shown, movements involving any pair of gates extracted from the set $\{1, 8, 7, 4, 9\}$ are not allowed, since they cannot decrease the number of tracks required by gate 2. Likewise, movements between gates belonging to the set $\{3, 6, 5\}$ are also prohibited. If the movement does not affect the critical column, any reduction in the number of required tracks will be only a local improvement, not global. For instance, suppose the gates 8 and 9 are swapped in Fig. 5. The number of tracks needed to implement the gates 8, 7, 4 and 9 might be changed, but the value corresponding to the critical column will continue to be six, making the individual require at least the same number of tracks.

The reduction in the number of swap and insertion tries when the CCBLS is utilized is considerable. In the example described in Fig. 18.5, the number of possible swaps and insertions without any reduction is 108. With the CCBLS reduc-
tion scheme only, this number drops to 49. The case in which there is more than one critical column is analogous. Since the movements must affect all critical columns, the prohibited movements are the ones in which both columns belong to the following regions:

- Before the leftmost critical column.
- After the rightmost critical column.

As said before, since the computational complexity of the local search neighborhoods is $O(n^2)$, the reduction will become particularly more appealing in the larger instances. Concerning the MA performance in practice, the reduction strongly improved it, making the algorithm reach better values with a much lower number of individual’s evaluations. The use of critical columns information has also allowed an increase in the number of nearest neighbors to be tested – the $k$-value. In other words, as the use of the CCBLS concentrated the computational effort only on movements that could improve the individual’s fitness, it opened up space to increase the search horizon while maintaining the computational complexity at a low level.

After a well-suited local search is developed, the next step is to determine how to better use it. There are several possibilities, such as applying the local search operator in:

- All new individuals created during the generation.
- A percentage of the new individuals.
- The best individual of the population.

The first possibility is the most natural, since the offspring generally evolve in equal conditions. The problem is that depending on the complexity of the individual evaluation, the number of local searches becomes prohibitively large. In fact, we dropped this possibility after the first test, where the algorithm took more than five minutes to execute just one generation using the 141-gate instance. This might be a good example where distributed processing could be successfully employed, since the local search in an independent process.

The use of local search in a percentage of the new individuals appeared to be more realistic. Nevertheless, we had the same problem of wasting too much computational time applying local searches in individuals that did not return satisfactory results. The results were still disappointing, regardless of the percentage utilized (tests ranged from 10% to 90% of the new individuals, with 10% steps).

The very best results came when we concentrated the local search on the best individual only. There is only one local search per evolutionary cycle\(^{11}\) and moreover, it is always concentrated on the most promising individual, what increases the chances to find higher quality solutions in a shorter CPU time. However, we should emphasize that this choice was strongly influenced by the problem structure, as well as the fitness function. It is not recommended to make a direct relation between the local search complexity and its level of application into the indi-

\(^{11}\) An evolutionary cycle begins with the initial population creation and ends when it converges to a low-diversity state.
individuals. This is a much more complex issue, being always necessary thorough tests considering a broad set of possibilities.

The use of the two neighborhood definitions – swap and insertion – together with the two reduction schemes, contributed to obtain a general superior performance, continuously surpassing the best previous approach to the problem\footnote{The best previous results for the GMLP were obtained by (Linhares et al. 1999).}. Moreover, the use of local search only in the best individual also helped to reduce the number of fitness evaluations, a second criterion utilized to compare the results.

### 18.3.5 Selection for recombination

The selection for recombination is one of the two processes that play the role of natural selection in the MA – the other one is the offspring insertion. There are many ways of selecting parents to give birth to an offspring. Among them, we must cite:

- Uniformly at random selection.
- Biased random selection (e.g., roulette wheel or tournament selections).
- Restrictive selection operators are usually used in structured populations (or breed-separated populations) or when there is a limit in the number of recombinations a given individual can take part in.

The unbiased random selection was the first approach utilized by a GA, in the original algorithm proposed by Holland (Holland 1975). It is very simple to be implemented, but lacks an important nature-related feature. Nature always privileges the best-fitted individuals to generate descendants. The reason is that such individuals tend to live longer, having a more numerous offspring than worse fitted ones. The biased random selections are successful in simulating such conditions. For instance, the roulette wheel approach selects parents with a frequency that is proportional to their fitness, meaning that the best individuals will have their genetic information transmitted to a larger number of descendants. This is a more realistic scenario, and usually leads to better results compared to the first option.

The restrictive selection operators are better suited for special reproduction scenarios. The tree-structure separates the population in different clusters and reproduction occurs only between individuals belonging the same cluster. In contrast with more static schemes, our clustered environment is more flexible, since individuals from different clusters can swap places if one becomes better than the other. This adds a different dynamics to the population.

In the hierarchically tree-structured population, the reproduction algorithm can only involve a leader and one of its supporters within the same cluster. The recombination procedure selects any leader uniformly at random and then chooses – also uniformly at random – one of the three supporters. The adoption of a uniformly-at-random cluster selection is related to the ‘multiple population’ character
of the hierarchical tree approach. By choosing the clusters in an unbiased fashion, we put the same ‘evolutionary pressure’ in all clusters. The use of biased selection operators is not necessary, since the evolutionary pressure on the best individuals is carried out by the population hierarchy adopted and by the offspring-acceptance policy. After an individual was created and went through – or not – a mutation process, the algorithm decides whether or not it should be discarded. This phase is described next, in Sect. 18.3.6.

18.3.6 Offspring insertion

The offspring insertion into the population is the second mechanism that plays the role of natural selection in the MA. Among the several existing insertion policies, at least three should be highlighted:

• Always accept the offspring.
• Accept the offspring with a probability that is proportional to its fitness.
• Accept the offspring only if it is better fitted than one of the parents.

The always accept policy has both strong and weak points. The strong one is that diversity is preserved for much longer and premature convergence is not a problem. The weak side is that since the population has a fixed size and all offspring are accepted, a situation where a good individual is replaced by a worse one might be possible, and even common. This could be partially overcome by setting the crossover rate at low levels and replacing individuals sequentially from the worse to the best one, in an elitist fashion.

The second policy is a more realistic representation of what happens in nature. In many species, the offspring has very limited parental support to survive its early days. The image of thousands of little turtles swarming over the beach heading to the sea just after being born is a somewhat suitable example. Predators are everywhere, choosing their preys among literally thousands of possibilities, in a strongly randomized process. The turtles that shall survive are the best fitted to reach the water as soon as possible and, of course, the lucky ones. The policy of accepting an offspring with a probability proportional to its fitness represents exactly this situation and shall improve the algorithm’s performance in comparison to the first one.

The third policy, which was utilized in the MA tests, states that the offspring will only be inserted if it is better than one of its parents, replacing the leader of the supporter that took part in the recombination. If the new individual is better than the leader of the cluster, it takes the leader’s place; otherwise it takes the supporter’s. This is an extremely elitist and restrictive policy, which generates a very fast loss of diversity. The positive side is that the algorithm becomes more “focused” early on and evolves faster. This characteristic was especially beneficial when no local search was employed. Although our tests are concentrated in the MA, we decided to make a few tests with a population-structured GA version, i.e.,

---

13 The first mechanism is the selection for recombination, described in Sect. 18.3.5.
the same MA without the use of local search. The use of this third policy strongly improved the performance of the algorithm, compared to the use of the two previous ones. This improvement was also noticeable in the MA, but with a lower intensity.

In order to deal with the accelerated loss of diversity, a more sensitive population-convergence checking had to be developed. Generally, population convergence is evaluated by the similarity degree of the individuals’ chromosomes and/or fitness. In this work we adopted a criterion stating that if during an entire generation no individual was accepted for insertion, we conclude that the population has converged. In such case, the heavy mutation procedure is applied thereafter.

During the recombination phase, after each generation, the population goes through a restructuring procedure. The hierarchy described in Sect. 18.3.1 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. Following this rule, the higher clusters will have leaders with better fitness and the best solution will be the leader of the root cluster. 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 GMLP, the higher is the position that an individual occupies in the tree, the fewer is the number of tracks it requires.

18.3.7 Pseudo-code of the MA

The Fig. 18.6 shows the pseudo-code of the MA described in the previous sections. The internal repeat-loop is responsible for the population evolution. Just before it starts, there is an initialization step, where all individuals are created and evaluated. Then, the processing enters the recombination for-loop itself, where initially, parents are selected and an offspring is created from these parents. The offspring is then mutated, evaluated and inserted into the population. This process is repeated until the last offspring is created. The population is then restructured and, if a convergence criterion is not satisfied, the process continues, otherwise the processing exits the internal repeat-loop. After the internal loop finishes, the local search is applied on the best individual and another population cycle begins. After all the populations were processed, the individual-migration phase takes place and afterwards, the restart of all populations. This process is repeated until a time limit is reached and the best solution ever found is reported.

Method MultiPopMemeticAlg;
begin
repeat
    for \(i = 1\) to numberOfPopulations do
        initializePopulations(pop(i));
        evaluatePopFitness(pop(i));
        restructurePop(pop(i));
        repeat
for $j = 1$ to $numberOfRecombinations$ do
    selectParents(individualA, individualB) $\subseteq$ pop(i);
    newInd = recombine(individualA, individualB);
    if (makeMutation newInd) then newInd = mutation(newInd);
    evaluateIndFitness(newInd);
    insertIntoPop(newInd, pop(i));
end
restructurePop(pop(i));
until (populationHasConverged pop(i));
end
for $i = 1$ to $numberOfPopulations$ do
    makeMigration(pop(i));
end
until (stopCriterion);
end

Fig. 18.6. Diagram of the implemented MA’s pseudo-code

18.3.8 Migration policies

The MA uses an island-type migration model. Consider several populations evolving separately in parallel, with little or no migration at all between them. Due to the genetic drift, explained in the Sect. 18.1, it is reasonable to conclude that after several generations, the final populations will differ a lot. This differentiation can be very useful for the MA search process. Since the populations will be composed of high quality, different individuals, the migration can create a precious synergy, leading the algorithm to solutions otherwise unreachable (Gordon and Whitley 1993; Levine 1994).

The island model needs a definition of a communication topology between islands, according to which individuals will migrate. One of the most studied and used topologies is the so-called ring structure. The populations are located at nodes over a ring, restricting migration to adjacent populations only. This approach apparently works better than the topology where all populations are connected. This second choice reduces the action of the genetic drift, worsening overall performance.

The next factor to be defined is the migration rate. It determines how many individuals will migrate after the population convergence. This is also a critical parameter since a too high migration rate might influence the genetic drift effect in the same way the topology does. We defined three options for it.

- **0-Migrate**: No migration is used and all populations evolve in parallel without any kind of individual exchange.
- **1-Migrate**: Migration occurs in all populations and a copy of the best individual from 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**: Migration does also occur in all populations, but two copies of each one’s best individual migrate to the adjacent populations, replacing randomly chosen individuals – except the best one. Every population receives two new individuals.

![Fig. 18.7. The figure displays the diagrams of the 1-Migrate and 2-Migrate policies in an example with four populations arranged in a ring structure. The arrows indicate migratory movements.](https://example.com/fig18.7)

The Fig. 18.7 shows the diagrams of two migration policies. The four populations are placed in a ring structure. In **1-Migrate** we have only one external individual being received by each population. In **2-Migrate**, this number rises to two individuals. The migration phase occurs always after all populations have converged and comparing the three policies is, in fact, a comparison between none, weak and strong migration, given the difference of communication intensity.

The last step is to determine the individuals to migrate. There are at least two choices: a randomly selected individual or the best one. Preliminary tests indicated that selecting a random individual is worse. The population’s best individual carries the best genetic information available and shall be more valuable in a future recombination with individuals from another population. Therefore, the migration was set to exchange only the best individuals among the populations. After migration occurs, all populations will go through a heavy mutation process that will restart them, leaving only the best individuals untouched, including the ones that were migrated. Therefore, after the migration and heavy mutation phases, the conformation of the populations will be as follows:

- In the **0-Migrate** every population will restart from the heavy mutation with just one high quality individual - the best one before the heavy mutation process.
- In the **1-Migrate** policy, every population will restart with two high quality individuals. The original best individual and the one received from the adjacent population.
- In the **2-Migrate**, the number of high quality individuals jumps to three. The original one and two received from the neighbor populations.
18.4 Computational experiments

The computational tests covered two aspects of the migration policies. The first was the influence of the number of populations on the overall performance. For this evaluation the number of populations varied from one up to five. The second aspect was the influence of migration intensity on the algorithm’s performance: for each number of populations, the three migration policies were tested, totaling 13 configurations – with only one population it’s not necessary to test 1-Migrate and 2-Migrate. The tests were applied into five instances, for which we tested the whole set of configurations, ten times each one. In Table 18.1 we show some information about the instances utilized in this work.

<table>
<thead>
<tr>
<th>Instance name</th>
<th>Number of gates</th>
<th>Number of nets</th>
<th>Best known solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>33</td>
<td>48</td>
<td>14</td>
</tr>
<tr>
<td>V4470</td>
<td>47</td>
<td>37</td>
<td>9</td>
</tr>
<tr>
<td>X0</td>
<td>48</td>
<td>40</td>
<td>11</td>
</tr>
<tr>
<td>W3</td>
<td>70</td>
<td>84</td>
<td>18</td>
</tr>
<tr>
<td>W4</td>
<td>141</td>
<td>202</td>
<td>27</td>
</tr>
</tbody>
</table>

There is one small instance, three medium-sized, and a large one. The work with the most extensive tests available in the literature (Linhares et al. 1999) presented 25 instances in total. However, most of them were too small and easy to solve either to optimality or to the best-known solution with the MA. Considering the instances’ sizes, only V4470, X0, W2, W3 and W4 had more than 30 gates and for this reason we concentrated the study on them. The stop criterion for the MA was a time limit, fixed as follows:

- 10 seconds for the instance W2.
- 30 seconds for the instances V4470 and X0.
- 90 seconds for the instance W3.
- 40 minutes for the instance W4.

The difference between CPU times is due to the dimension of the instances and takes into account the average time to find high quality solutions. The increase in the CPU time from W3 to W4 reflects an explosive growth in the computational complexity. Although the instance size has just doubled, the time required to solve it was multiplied by more than 25. All tests were executed in a Pentium 366 MHz Celeron\textsuperscript{14} computer, using Sun JDK 2.0 Java language running under Windows 98\textsuperscript{15} environment. For comparison reasons, we estimate that the Pentium Celeron processor’s performance lies somewhere between a Pentium and a Pentium II processor. The Java 2.0 is a just-in-time native compiler, much faster than previous Java versions.

\textsuperscript{14} Pentium and Celeron are trademarks of Intel Microsystems.
\textsuperscript{15} Windows is a trademark of Microsoft Inc.
The experimental results are shown next. Four figures describe the results for each configuration (see Fig. 18.8). In clockwise order we have in boldface the best number of tracks found for that instance. Next in the sequence we display the number of times this solution was found out of ten tries, the worst value found, and finally, in the lower-left part of the cell, is the average number of tracks.

![Diagram](image)

**Fig. 18.8.** Data fields for each configuration of number of populations and migration type

The instance W2 has reached the presumed optimal value of 14 tracks in all configurations of migration policy and number of populations. However, we could still conclude that the best configuration was the 1-Migrate with four populations because it required the lowest number of individual’s evaluations. The other instances have shown more noticeable differences in performance and their results were thus detailed in the Tables 18.2 to 18.5.

Since the tests take into account two parameters – migration policy and number of populations – we will analyze them separately. At first, two aspects of randomized search algorithms must be pointed out: exploitation and exploration. Exploitation is the property of the algorithm to thoroughly explore a specific region of the search space, looking for any improvement in the current best available solution(s). Exploration is the property to explore wide portions of the search space, looking for promising regions, where exploitation procedures should be employed. In general, the island model of migration privileges exploration, with each population searching in a different part of the solution’s space. Moreover, when an individual is migrated, the intention is to explore the region between the ones covered by the populations. The exploitation job, carried out mainly by the local search operator, is thus simultaneous with the exploration.

<table>
<thead>
<tr>
<th>Table 18.2. Results for the V4470 instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of populations</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0-Migrate</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1-Migrate</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2-Migrate</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
The first parameter – migration policy – has a trifling influence on the algorithm’s performance. Nevertheless, the 1-Migrate appears to be the logical choice, since the best configuration for each instance utilized it. The other two policies have also had a good performance but were slightly worse. The 1-Migrate apparently better-balanced exploitation and exploration, returning the W4 presumed optimal solution – 27 tracks – in two different configurations. The other instance where migration has impacted the results was the V4470, which revealed to be a rather difficult problem. The 0-Migrate option was clearly worse than the other two, on average.

The second aspect to be analyzed is the number of populations. Although it is also not clear which configuration was the best, the use of only one is surely not the best choice since several multi-population configurations gave better final values. Based on the results, the conclusion is that when multiple populations are utilized, at least three of them should be employed. With only two populations, the algorithm does not seem to take advantage of the genetic drift effect. The use of
more populations allows the exploration of much larger portions of the search space, being necessary at least three of them to make this feature noticeable.

The other characteristic of the method was its high efficiency. The number of individual’s evaluations was very low. Compared to the best previous work (Linhares et al. 1999), the improvement is striking. Initially, we shall briefly explain that approach. The Microcanonical Optimization (MO) presented in (Linhares et al. 1999) is based on a fast variant of the well-known Simulated Annealing (SA) (Kirkpatrick et al. 1983) approach, which divides the search into two alternating phases, initiation and sampling. These phases have dual objectives: in initiation, the system strives to rapidly obtain a new locally optimum solution, while at the sampling phase the system moves out of the local optimum while retaining similar cost values (as controlled by parameters analogous to the temperature in SA). The proposed algorithm was able to outperform five previous approaches in all previously tested instances. The MO is also fully described in (Linhares et al. 1999). The Table 18.6 shows the results for both methods.

**Table 18.6.** Number of individual’s evaluations required by the memetic algorithm and by the microcanonical optimization approaches to reach the presumed optimal solutions

<table>
<thead>
<tr>
<th></th>
<th>Memetic Algorithm</th>
<th>Microcanonical Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
<td>Average</td>
</tr>
<tr>
<td>W2</td>
<td>3,125</td>
<td>3,523</td>
</tr>
<tr>
<td>V4470</td>
<td>32,509</td>
<td>176,631</td>
</tr>
<tr>
<td>X0</td>
<td>18,136</td>
<td>43,033</td>
</tr>
<tr>
<td>W3</td>
<td>79,089</td>
<td>203,892</td>
</tr>
<tr>
<td>W4</td>
<td>3,213,532</td>
<td>9,428,591</td>
</tr>
</tbody>
</table>

Analyzing the results presented in Table 18.6, we conclude that the MA obtained much better values than the MO approach, in all instances. Taking the average results, the reduction is better than 80%, depending on the instance size. For the largest instance, W4, we had an outstanding 94% average reduction in the number of evaluations. Such results were absolutely unexpected, and thus we decided to check for any bugs in the software code. After discarding this possibility, we ran both algorithms – MA and MO – on the same machine. The result was finally confirmed by the difference of CPU time. For instance, the MO required over six hours of CPU time to find the W4 optimal value, while the MA found it within less than 20 minutes.

The comparison of the number of evaluations has also brought some light into the V4470 instance results. Although the V4470 is smaller than the X0, the MA had some difficulties in finding the optimal value. This was reflected in the number of evaluations, much larger than for the X0 instance. Initially, we thought this was a problem of the MA, but after checking the MO results we concluded this instance is intrinsically difficult, deserving a more careful study on its structure.

There are at least three reasons for these outstanding results. First, the local search embedded on the MA does not consider all possible movements, as a neighborhood reduction enables considerable computational gains (without major
loss of search power). Second, the idea of discarding all possible movements that do not affect critical columns (and hence cannot improve solution quality) also is a major step towards greater efficiency. Finally, the application of local search is carried out only to the best individual of each population. These factors, taken in combination, help to explain the superior performance of this algorithm. Of course, all other factors – migration policy, crossover strategy, mutation, population structure and offspring’s insertion policy – also played an important role. Nevertheless, the local search is by far, the most performance-influencing feature of the MA. Now, in order to give more information about the speed of the algorithm, the Table 18.7 presents some MA-related statistics.

Table 18.7. Statistics on the MA related to the number of individual’s evaluations – the figures represent average values

<table>
<thead>
<tr>
<th>Instance name</th>
<th>$k$-value</th>
<th>Evaluations per local search</th>
<th>Evaluations per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>10</td>
<td>379</td>
<td>6,906</td>
</tr>
<tr>
<td>V4470</td>
<td>20</td>
<td>548</td>
<td>5,702</td>
</tr>
<tr>
<td>X0</td>
<td>20</td>
<td>585</td>
<td>5,557</td>
</tr>
<tr>
<td>W3</td>
<td>30</td>
<td>749</td>
<td>3,374</td>
</tr>
<tr>
<td>W4</td>
<td>60</td>
<td>3,242</td>
<td>662</td>
</tr>
</tbody>
</table>

The $k$-values presented in Table 18.7 are the same used in the computational tests. Next to them we have the number of evaluations per local search. This number does already take into account the two local search reductions – the $k$-based limit and the CCBLS. Finally, the fourth column shows the number of evaluations per second. The evaluation procedure requires the entire 0-1 matrix to be scanned and its complexity order is $O(g.n)$. There is little space for optimization and thus two different implementations of that procedure cannot differ too much in terms of complexity. We believe this value might be a good performance measure between different computer systems, facilitating future comparisons.

18.5 Discussion

This chapter presented a memetic algorithm (MA) application to the gate matrix layout problem (GMLP), which belongs to the NP-hard class. The main features of the MA were described, including:

- A hierarchically structured population and its specific selection for recombination procedure.
- The BOX crossover – a variant of the well-known OX crossover.
- The local search and two neighborhood-reduction policies.
- An island-model migration.

The items cited above were critical for the better performance of the MA, compared to the previous best method for the GMLP. Many questions might arise at the end of this chapter. Is the ternary-tree structure really the best choice? Can the
local search be improved even further? Should the ring structure of the island-model be replaced by another one?

Every time a general method is specified and used on a given problem, questions related to the finally specification naturally start to pop-up. Nevertheless, the main goal of this implementation was to join new ideas, like the hierarchical population, the BOX crossover and the CCBLS local search, with “consecrated” ideas, such the island-model migration, with the intention to solve a complex combinatorial problem.

A conclusion derived from this implementation, which is worth emphasizing, is that local search is very important. No matter how hard is the problem being solved, if a GA was already developed for it, it seems reasonable to attempt a MA adding an ad hoc local search procedure to the GA.

The use of a standard set of “industrial-sized” real-world instances was very important in our study. Their sizes varied from 33 up to 141 gates. The solutions obtained have rivaled with the ones from previous methods’, matching all the best-known solutions, while dramatically decreasing the computational effort. In all cases, the number of evaluations was reduced by at least a factor of eight.

This algorithm is included in a framework for general optimization, called NP-Opt (Mendes et al. 2001). This framework is an object-oriented, Java-based software. At present, it includes five different NP-problems: Single Machine Scheduling, Parallel Machine Scheduling, Flowshop Scheduling, Gate Matrix Layout and Gene Ordering. The framework is updated and improved continuously by a team of collaborators. For more information, please refer to the NP-Opt Homepage\(^{16}\), where the latest version is always available for download, as well as the software guide and a set of test instances. This guide describes technical aspects of the framework including the classes’ structure. It will also help the user to change the software code, adjusting it to different needs. More complex issues like how to add new methods, problems, and changing the graphical interfaces are also discussed in the user manual.

Although the MA used to address the GMLP contains several specially tailored features, it runs in this general optimization environment. The use of the NP-Opt framework facilitates the programming of optimization methods. We believe that further development and study of such frameworks may ultimately provide an invaluable tool to the field of systems science.

\(^{16}\)http://www.densis.fee.unicamp.br/~smendes/NP-Opt.html
References


18. VLSI design: gate matrix layout problem


