Solving NP-problems using genetic algorithms

Bruno Böttcher

01.01.2018
original source 12.1993 - 6.1994
Version 0002.12h06

Abstract

This paper now goes on with the android version of a utility framework for solving NP-complete problems using a genetic approach.

After a introduction about the history and problem setting of the project, we go on discuss how a possible implementation of a genetic algorithm could be and its associated application.

This framework could be/was used to be applied to varying sets of setups going from simple toys as a image puzzle solver up to part of a market simulator.

This port of the C++ framework to Java and Android is still rudimentary, supporting the project through buying the app will ensure it continuation.

Contents

Chapter 1

Introduction

The present work builds upon my thesis (diploma) of electrotechnics in 1994.

The starting problem was a NP-problem [wikiNP], a nondeterministic polynomial complete problem, meaning the sort of problem where you can't test all possible cases in finite time, but checking the viability of a given solution can be computed in a time growing in polynomial way versus its size.

Examples for NP-complete problems are the traveling salesman, or in the case of my thesis floor-planning, where different elements of varying or fixed size had to be placed using a minimum amount of space, or taking into account some extra constraints, as the number of connections between the elements and their length.

In my thesis a number of solving algorithms were reviewed genetic algorithm was selected because of the inherent parallelity and thereby the expected computing time profits using distributed systems, in comparison to the rather long computing times using sequential processing.

Later then, and since i already had a working implementation, this sort of solving algorithm was used in a varying set of setups going from simple toys as a image puzzle solver up to part of a market simulator.

As time evolved, the program was ported from its implementation in C++, to Java and now finally making its first steps into the Android world.

This paper now goes on with the android version of this utility framework.

The first class of solving problem that was tried to solve with the original framework treated placing algorithms. Placing algorithms are used for the automated draft of VLSI-circuits. VLSI-circuits are one of the today's base technologies: the implementation of complex electrical circuits on smallest space. VLSI-products gain access in many areas like communication, medicine, automobile industry etc.. With growing requirements on functionality and the progress in micro-electronics the complexity of manufactured circuits explodes. The draft of circuits with several thousand transistors on a chip in economicly meaningful time is only possible with automated aids.

The draft of VLSI–circuits is usually partitioned in several steps :

- The first step is the **logical draft** of the circuit, taking the problem definition, a hierarchical block diagram is created.

- Subsequently, the **physical draft** (*Layout*) follows:
  - Placing: In the Layout-phase the placing is the most important part, determining if a solution can be found for the routing-process. That is why the development of more efficient *placing algorithms* is one of the principal aims that animate Draft automation. Placing itself can be partitioned in:
    - **Floor-planning**: try to find a favorable placing of the modules with a reduced set of design–criteria. For example, criteria like the non–overlapping of cells or technological restrictions (e.g. row arrangement with Gate–Array design) are firstly not taken into account.
    - The **final placing** produce the conformity to the skipped less important criteria (e.g. non overlapping).
- The **wiring**, "*Routing*", tries to place the connections (wires) between the placed cells.

  Since it is not possible to guarantee a solution of the routing process, placement and routing are iteratively executed until a solution is found.

Whilst the included example is a simplified down version of the C++/Java-placing algorithm, it is interesting to note that it can be used in nearly identical way when:

- a puzzle is solved: each piece knows which which pieces it is connected, making a group of tiles who have to be ordered in a way to minimize the distance between neighbors, or increase the scale,
- a county, where through inheritance, selling and reselling, the land was fractioned between a multitude of persons, and now the need to restructuring comes, where all parcels pertaining to one person should be regrouped taking into consideration ease of access, quality of the soil, access to irrigation, etc.
- a market has to be analyzed where multiple agencies, who have different links one to another have to evolve in a way to perhaps grasp future trends.

The problem of those algorithms lies in the fact that usually several optimization criteria must be considered at the same time. To compliquate the problem the different criteria often negate each other. E.g. even in the simplest case of wire length minimization the problem belongs to the class of the *NP complete* ones; this means that the computing time exponentially increases with circuit complexity, making impossible the search for a strictly optimal solution with deterministic procedures. Therefore *heuristic procedures* are applied.

The heuristic procedure was chosen in analogy to a biological natural phenomena: the evolution of a population (reproduction of the individuals with the best characteristics) "*Genetic*" .

For the sake of simplicity, and if not otherwise specified, we will use the case of VLSI cell placing as a base for our reasoning, but transferring said reasoning to the other brushed examples is rather trivial.

Chapter 2

Placing algorithms

2.1 General information

2.1.1 Input from the logic draft

The logical draft provides the following data:

- The list of the Modules M, described in a matrix $\mathcal{M}$, in witch together with the cells $m_i \in \mathcal{M}$ is included data about the cells, like the coordinates of the Cell, possible time-critical behavior, increased power dissipation and similar things, this supplementary data is called *Attributes*.
- The connection–list $\mathcal{N}$.

Example of five interconnected cells, their ideal arrangement can be seen in the Picture 2.1.

The extracted data for this example could be like the one showed in the following picture 2.2. Thus the attribute matrix $\mathcal{M}$ and the net-list $\mathcal{N}$ could have following aspect:

The attribute matrix contains beside existential data as wire length also different additional library data, like for example "Aspect– Ratio", used by the more complex algorithms to take better account of the form of the cells. Aspect–Ratio means the different side relations, a cell can take with different possible layouts. This

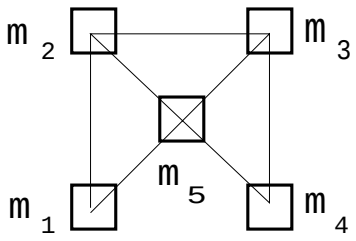

Figure 2.1: Five connected Cells

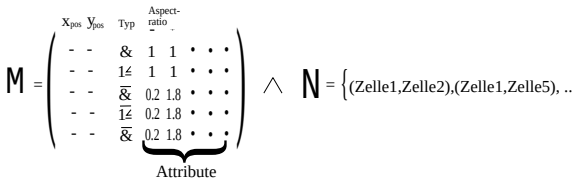

Figure 2.2: Input from the logical draft

follows from the fact that, depending on where the inputs-/outputs (pins) have to be situated, different cell–layouts exists to fit the best needed way.

In the matrix from the example the x– and y–positions are filled with dashes. In the case of a genetic solver, which is said to be of the class of iterative placing procedures, versus placing algorithms (not treated here) iteratively tries to improve an initial proposal. These need an initial placement, which can be gained from a random distribution or from the result of a iterative placing procedure.

2.1.2 General terms

In this work the word "*cell*" stands for the item to be placed. A cell is the smallest unit to be placed. This can be a transistor or a macro–cell[1]

To simply search for a solution that fits in a particular rectangle being a bit easy, we introduced the concept of linking the cells through wiring.

Since free wiring (curved) is very difficult to realize with automatic draft a limitation is introduced in form that the *router* only uses horizontal and vertical segments. In analogy to the special architecture of Manhattan this geometry type is called *Manhattan–geometry.*

Minimum wire length and half rectangle scope (LHR)

The restriction due to the Manhattan–geometry influence the wire length: the minimum distance between two pins corresponds now to the *half Rectangle scope* and no more to the Hypotenuse. Half rectangle scope means the sum of a vertical and horizontal edge of the Rectangle, which has the two analyzed pins as diagonal corner points:

$$L_{HR} = |x_{Pin1} - x_{Pin2}| + |y_{Pin1} - y_{Pin2}| \tag{2.1}$$

Since there is no shorter way with this restriction, the L_{HR} length describes the minimum wire length. For this reason the L_{HR}–length is used as a lower limit wire–length estimation. This estimation is valid only for networks with two to three pins. For networks with more pins an increasing error occurs. Here networks means the wires that connect the cells.

[1] a macro cell consists of a net of standard cells, these standard cells for their part are constructed with a net of transistors.

However statistically seen, in a circuit the networks with two to three pins outweigh, so a relatively good estimation can be made with the half Rectangle scope.

Thus the minimum wire length between two with n networks (= wires) connected cells becomes:

$$L_{min} \quad _{k,i} = \sum_{\nu=1}^{n} L_{HR} = \sum_{\nu=1}^{n} |x_k - x_i| + |y_k - y_i| \tag{2.2}$$

In the case of the Floor-planning generally for reasons of simplification all pins are placed into the center of the cell:

$$L_{mink,i} = \sum_{\nu=1}^{n} L_{HR} = n * L_{HR} \tag{2.3}$$

The entire minimum wire length of the placing proposals results from the sum of the individual wire lengths.

Connection weight

In some cases an appreciation of the number of connections (Networks) per cell was needed. The number of connections is called *Connection weight*. The Connection weight is defined as follows:

$$V_i = \sum_{k \in \mathcal{N}_i \cap \mathcal{N}_j} (\frac{n_k + \lambda}{n_k}) \quad \wedge \begin{cases} n_k = \text{Number of pins from} \quad \mathcal{N}_i \cap \mathcal{N}_j \\ \lambda \quad \text{usually =1} \end{cases} \tag{2.4}$$

The $\mathcal{N}_j$ and $\mathcal{N}_i$ describe the networks with origin in the cells j and i. Thus the cut of the two Network-lists gives the common connections of the two cells. λ is a correction factor for the adjustment of the function.

Cost function

With the preceding definitions a cost function can be defined:

$$K = \alpha_1 L_{min} + \alpha_2 F_{Chip} + \alpha_3 F_{ov} + \cdots \wedge \begin{cases} L_{min} & \text{minimal total wiring length} \\ F_{Chip} & \text{used chipsurface} \\ F_{ov} & \text{overlapping} \\ \vdots \end{cases} \tag{2.5}$$

The weight factors α accentuate the importance of certain attributes.

The sense of the cost function, (usually only the difference ΔK is used), consists of judging the quality of a draft. Depending of the placing algorithm more or less attributes are taken into account, whereby the minimum wiring length L_{min} the Basic structure forms. Theoretically the cost function can be arbitrarily defined. Here are defined the different priorities of the attributes. According to experience a linear cost function like the one in the Formula has shown best behavior.

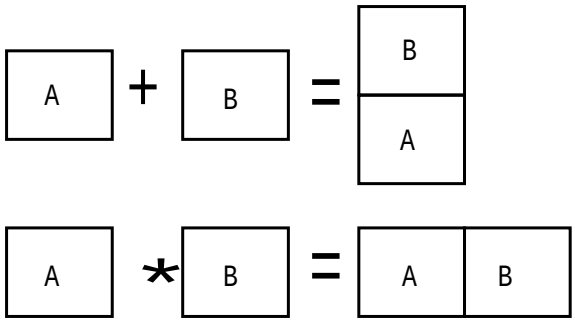

Figure 2.3: picture about the Polish Notation

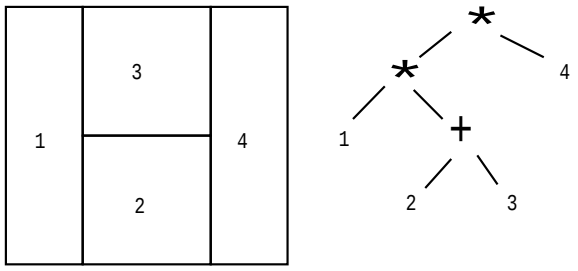

Figure 2.4: example of the translation of a placing proposal as slicing structure

2.1.3 Polish representation

The result of placing is a partitioning of the chip surface in subregions of the size of the cells. This fact can by described with the introduction of the cut tree "*slicing tree*". The placement does not anymore be described as a sequence of cell–coordinates, but as a sequence of vertical and horizontal cuts. For this a symbolic representation is needed: a "*" stands for a vertical and a "+" for a horizontal cut (see picture 2.3).

This representation was taken also as an analogy to the way proteins produced by a genetic DNA strand fold themselves.

Since a given partition can be described with different polish representations the normalized representation is used. Here the cuts are made alternatingly horizontally and vertically. So a hierarchical structure is developed, which can be represented by a binary tree. Each "leaf" corresponds to a cell and is provided with a number.

$$\mathcal{M} \quad = \quad \begin{pmatrix} 1 & 2 \\ 4 & 1 \\ 4 & 3 \\ 7 & 2 \end{pmatrix} \tag{2.6}$$

The matrix describes the placing shown in the picture 2.4. The coordinates describe the center of the cells to be placed.

In order to be able to handle this structure more easily with computers, the tree is read out. The result is an arithmetic expression in inverted polish representation.

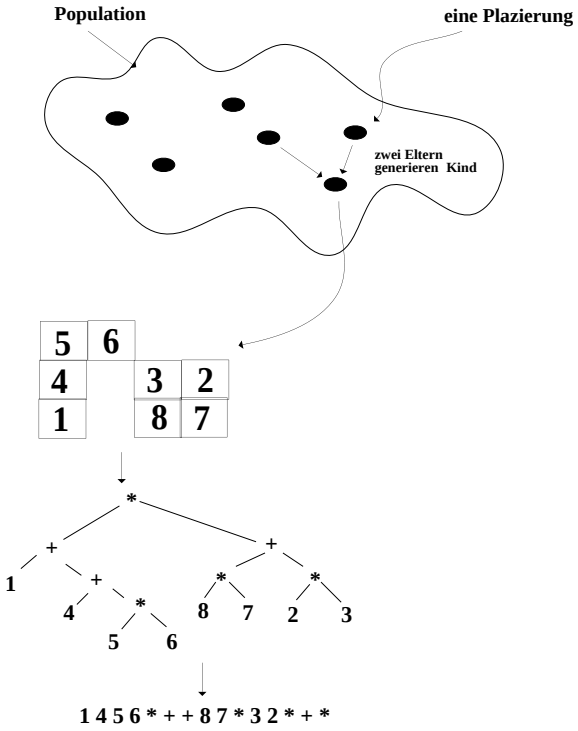

Figure 2.5: picture to a genetic Model

This means that the operators are placed before the operands. Since the cut graph can be read out in different manners the following convention is introduced: The cut graph is read from top to bottom and from the right to the left and this structure represented as a tree. The result is read vice versa, thus from bottom to top and left to the right. So a character string in Polish notation is achieved, which can be defined as normalized (see pictures 2.3 / 2.4 and Formula).

$$123 + *4* \tag{2.7}$$

2.2 Genetic placing algorithms (at the example GAPE)

Genetic algorithms are characterized by the following approach: at each time exists a solution set, called *population*, which is permitted to "procreate", to develop itself further (see picture 2.5). Single solutions are called "individuals". The next generation consists of the survivors of the preceding. The survivors are the *individuals*, with the best caracteristica (the *fittest*). A suitable representation must permit the judgment of the aptitude of the different solutions and the creation of new solutions, see [wikiGA] for a more in-depth discussion about genetic approaches to problem solving.

In analogy to [Wong86] and [Coho91] the description of the cut tree as polish

representation is selected. This representation has the advantage of making the partitioning–data available in a vector. Further it describes a hierarchical structure. The advantage stays in the fact that the lower sub–trees (which contains the *leaves*) represent the strongly interlaced subnetworks. This makes some operations easier:

- it relieves the new calculation of the cost function, here called "*fitness*";
- The structure of the proposal can be better displayed.

Thus a placement–proposal represented in polish notation is called in the subsequent text "*placement–rule*".

In the case of the reproduction two phases are differentiated:

- The generation of offsprings by two parents. The characteristics of the parents mixed and transferred to the descendant In senses of placing: The information, which is contained in the placement–rule of the parents is merged with the help of the so-called "*cross–over–operators*" into the offspring.
- The uncontrolled modification of the placement–rule by mutation.

2.2.1 The quality, aptitude or "*Fitness*" of a population

The first genetic algorithms used simple relations for the measure of the Fitness with the cost function (K):

- $\sim \frac{1}{K}$
- $\sim (C - K)$ where C is a constant, which must be big enough avoiding the Fitness to become negative.

In GAPE a better, dynamic formulation of the Fitness was selected. It is determined by the momentary cost function:

$$\text{Fitness}(x) = \frac{(\mu_K - K(x)) + \alpha\sigma_K}{2\alpha\sigma_K} \tag{2.8}$$

where:

α Weighting factor is, which is set initially to 1.

μ_K is the expectancy value of K

σ_K is the standard deviation of K

Further it is defined that, if the Fitness becomes smaller than zero it is set equal zero, since no negative Fitness should exist.

This follows the observation that in the case of random distribution the cost function obeys to a Gauss–distribution.

This is still a TODO for the android project, since actually only one cost–function is implemented, which takes into account deviance of the quadratic from of the solution, and the wiring cost.

Conversely, and since only on way of weighting the solution is implemented at the moment, the cost was taken directly to compare the fitness of the different solutions.

2.2.2 Mixture of the characteristics, *Cross–Over CO*

Here are introduced the Cross–over–operators from GAPE.

GAPE differentiates four different CO–operators, $CO_1 \cdots CO_4$

- CO_1/CO_2: are thought as character string manipulators, the result is again a normalized placement–rule. They produce out of two parents one descendant. Each time a complete placement rule is taken:
  - CO_1 copies all operands from P_1 on the same positions on the offspring O. That means that the operands of the offspring O are on the same places and of the same type as in the P_1. Subsequently, the operators are taken out of P_2, selected from left to right and inserted in the vacant positions of the offspring. Thereby the operands (= cells) which were clustered in P_1 remain together, whilst the overall cut–structure is changed.
  - CO_2 is inverted to CO_1. Here the cut structure P_1 is copied into the offspring O and the operands transferred into the vacant positions.
- CO_3/CO_4: These partition–rules achieve the transmission of subtrees. The increase of fitness occurs at the beginning in the subtrees. Therefore this two CO–operators are specialized to manipulate sub–trees:
  - CO_3 selects a sub–tree S (= character string from the partition–rule) out of P_1. A character string represents really a sub–tree, if the rightmost character is an operator, and the sum of the operators equal to the sum of operands minus one is. This subtree is inserted at the same place in the offspring O. This place is then excluded from the following steps. Now a operation like in CO_2 is performed: the remaining cut structure of P_1 (= the operators) is transferred on the offspring. The operands from P_2 are now filled in the vacant positions. Here must be taken care that no two equal cells are in the offspring placement–rule. Whilst the selection of the current operand out of P_2 a check must be performed for presence in the offspring in S. If this is test is positive, this operand is ignored and the next one taken.
  - CO_4 produces two offsprings: in both parents a subtree of equal dimensions is selected. Now the sub–trees are exchanged: two steps with CO_3 are executed: P_1 and S_2 are merged to O_1 and P_2 and S_1 are merged to O_2.

The Cross–over–operations are shown graphically in the picture 2.6.

In the actual implementation only crossover 1 to 3 is implemented, the 4th variant adding only programming complexity without really adding deeper solution space exploration.

The Cross–over–operators cause big jumps in the solution space at the beginning. With rising Fitness they cause however only an exploration around the fittest

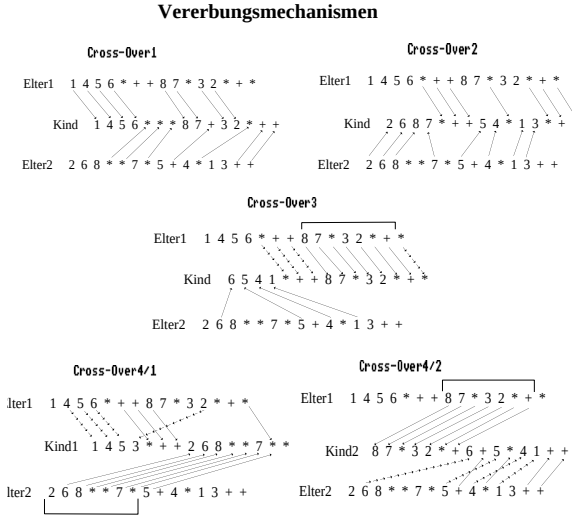

Figure 2.6: The Cross–Over operations

solutions. The modification of the behavior is not due to a change of the CO–operators, (in the Implementation of the algorithm they remain constant), but by the fact that the parents grow together. This approaching is called speciation or "*clustering*". Empirical studies showed best results for random selection of the four CO types at cross–over stage.

2.2.3 Mutation

Mutation selects randomly an individual and changes it, alternating one place in the partition–rule swapping two adjacent fields. In the original implementation, all combinations were permitted :

- two operands
- an operand and an operator
- two operators

The problem with this operation is that the partition–rules obtained could be not normalized. This is however permitted within original GAPE, since it was demonstrated that the quality of the solutions was thereby improved. If the Mutation really operates randomly, better results can be achieved since a bigger solution space is explored.

But the hassle to cope with non-normal trees, and thus the associated computational increase didn't show for my test runs the big benefit advertised, thus i dropped this feature, changing the mutation function to check:

- if a leave, then ask the parent node to exchange left to right

- if a node randomly change the type of the node
- for a future TODO allow to exchange nodes in a bigger subtree than just the parent node

At the start the Mutation produces big jumps in the solution–space, whereas only a very weak effect can be noted at the end. This is due to the fact that in the case of high fitness of the individuals a mutation produces a big increase of the cost function. Nevertheless the features of the bad solutions can be passed on, since the selection of the individuals for the next generation occurs only at the end of one generation. Thus parts of the new solutions which possibly lower the cost function can be used in conjunction with the fitter individuals. For this reason the number of mutations is not reduced at any moment.

But another concept was introduced, instead of strictly separating mutation and cross-over phases, it was chosen to mix them, allowing also certain cross-over offsprings to incur a mutation on the fly. At the moment , the probabilities are 75 For a further TODO, it would be nice to make those probabilities adjustable by the client, this being, as with with many other heuristics, one of the many parameters that want to be set. The right choice making the difference between a good solution and a bad one.

2.2.4 The basic concept of GAPE: "*Punctuated Equilibria PE*"

"*Punctuated Equilibria PE*" was developed as a proof for the darwinistic theories and proved in the nature. The predicted result is: if an organism is introduced in a new, empty and closed system, then it rapidly begins to develop itself in many subspecies ("*allopatric speciation*"), until the available ecological niches are filled. When the niches are filled, the different species get in to balance ("*Stasis*") and there is only a weak genetic Drift.

- *allopatric speciation* means the rapid formation of new Genotypes (species) from a small genetic set (for example the birds on the Galapagos–islands).
- *Stasis* is reached, when each ecological niche is filled. Mutated individuums are not able to survive. The available resources are occupied by the existing kinds. This brings to predicate following rule: a kind of being survives, as long as does its environment.

Thus it can be said that a good method for bringing up new kinds out of a given set, consists of bringing those well-known kinds into new environments. Genetical algorithms (GA) produce a set of similar solutions (types). These form a stable system, which hardly changes (Stasis). Thus it can occur that the solution space in the proximity of the minima is only insufficiently investigated. This fact predicts better results for GAPE than for GA. GAPE can be described as making a parallel

run of many singular genetic algorithms, in which at regular intervals happens a catastrophes to force new development.

The original [Coho91] refers that a parallelization of genetic algorithms is possible by distributing the Cross–Over and Mutation operations. This would result in an "*Hardware–accelerator*". They present a method, which promises a higher gain in speed. Therefor a new concept is introduced: **punctuated Equilibria**.

In order to work efficiently, this algorithm needs several closed locations, where different populations can grow up. This step is suitable for parallelization: Each processor receives a population of m Individuals. In parallel are executed the generations and at the end of one "Epoch"(=defined number of generations; theoretically, reaching Stasis) each processor sends its fittest individuals to the other processors. If a fixed Epoch–length is used, the synchronization of the processors is simplified since the load can be set in order to let all processors terminate at same time.

This way the fittest solutions from all the areas must go in concurrence against each other. With the partitioning of the evolution–process a bigger area of the solution space is examined. The Fitness calculates in dependency of the local–populations. Thus modifies the estimation of individuals with introduction of new ones. With the number of exchanged individuals after each epoch, the range of the catastrophe can be adjusted. Further it would be conceivable to put different accents on attributes in the cost function of the different environments.

The algorithm is from the complexity $O(m^2)$, determined by calculation of the cost function. The authors [Coho91] state that a linear speed increase is achieved along with the number of processors. This is supported by [Bril90].

For this reason and the inherent parallelity this algorithm was selected for implementation.

A further reason to prefer GAPE–like algorithm is situated in the attainable quality: the analytic Algorithm provides after a calculable number of cycles only an almost optimal result. The genetic algorithm can with favorable adjustments of its parameters, along with sufficiently computing power and time deliver optimal results.

The problem with this is situated in the correct adjustment of the parameters. Here a big experience with placing problems generally and genetic algorithms specially is necessary.

In the Java-gape project, this was achieved with every client holding a population of individuals working each with another cost-function. And, indeed, a wider search of the solution-space was noted, even if the total number of individuals per population had a bigger impact on breaking the punctuated equilibria problem.

The Android Gape still lacks support of multiple populations. Furthermore, the different environmentswëre simulated through different cost-functions, another feature not yet present in the android version.

Still, the sending and receiving of ïndividualsïs implemented, a system using multiple populations, and working distributedly over several android devices wouldn't be too hard to implement at the actual stage.

Chapter 3

Program environment

3.1 Android specific problems

Since the life-cycle of an Android app depends on the available RAM and processing power of the device on older devices, and depending on the size of the chosen example, it appeared that the program is killed as soon as it leaves the actual context, thus making it foreseeable that if several devices want to cooperate for a given solution the program probably will need to stay in the foreground, also due to the fact that the use of heavy duty background services are not in the favor of actual google-app-policy.

3.2 Definition of a test–case

A standard case was defined: meshed network of homogeneous linked cells is build, at the moment only one of the mesh–factories is implemented in the android-version, building a varying sized mesh of cells linked with their neighbors, in matrix form. In this case the optimal result is easily conceivable for a e.g. 10x10, see picture 3.1.

3.3 Results of computation

A representative number of configurations still need to be run, so far we can offer: nevertheless a few statements could be won:

- too short epochs with few populations cause rapid homogenization of the populations. This produces rapidly identical populations, this leads with high probability to a local minimum and no longer corresponds to the concept of *punctuated equilibria*.
- more individuals per population seem to bring no improvement on the result, however extending the number of populations. Here must be found the right equilibria between size of the problem, number of populations and number of individuals per population.

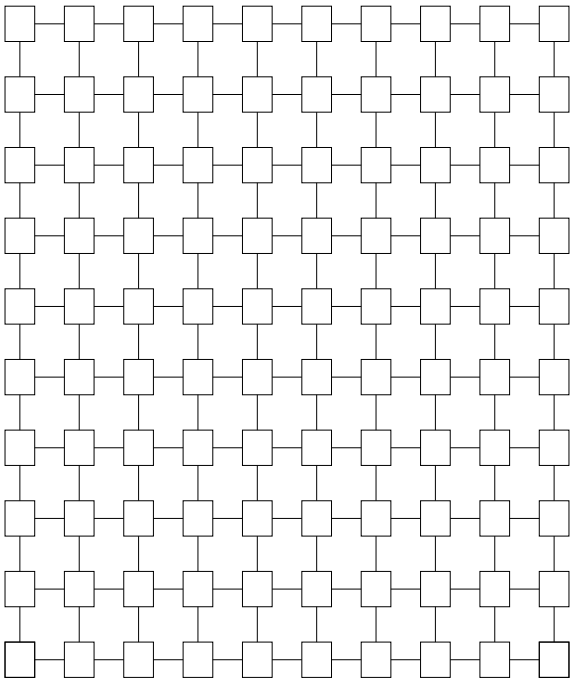

Figure 3.1: Homogeneously interlaced Modules

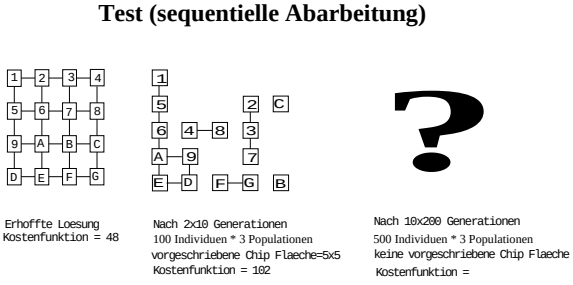

Figure 3.2: calculation–examples

- defining a surface limit factor into the cost function has shown positive effects, preferring the solutions that fit on the reserved surface.
- as very important emerged the fact that no identical solutions should be allowed. Otherwise the population fills rapidly with one good solution that emerged. This effect is more important in our implementation since we use the sorted array, and when the effect of duplication begins, it rapidly take the whole population over.

Chapter 4

Summary

4.1 Summary

This project achieved only part of what the C++ original was able of, at the moment the basic genetic approach was implemented, the parallelization, and the ability to stack hierarchically different solving steps (solving the global problem, soling at the same time, the placement also of the subproblems inside the cells) wasn't implemented.

In the case that unexpectedly some funds are raised, the android version would be developed to match at least the state of the 1994 version, with multiple populations/cost-functions and distributed clients.

4.2 Next objectives

- The existing statistical functions of the algorithm are thought to help observe the evolution of the populations. Precisely the process of mixing populations should be examined more closely: The influence of bringing individuals into an existing population, regarding to the difference of fitness and related to the number of invading individuals versus the population size.
- allow the sending of custom cost–functions by the users, to allow different interpretations of the found solutions. This is already prepared, but not fully implemented/tested the floor-planning module being the only one implemented.
- with more incentive i could port the puzzle solver, which cuts up a photograph, puts it on top of a mesh, scrambles the mesh, and tries to find the image back.

Appendix A

Used Symbols

$\mathcal{M}, \mathcal{N}$	Matrices
m	Cell = element of $\mathcal{M}$
$x_{...}$ $y_{...}$	Coordinates
$\mathbf{L}_{HR}$	Half Rectangle scope
$\mathbf{L}_{min\quad k,i}$	minimal wiring–length between Cell k $\wedge$ i
k, i	indices, used to differentiate two cells
V_i	connection–weight
λ, α	Correction–coefficients
$F_{...}$	surfaces, areas
K	Cost–function
T	Temperature
Φ	target–function
$g_{\mu\nu}$	weighting factors
μ_k	expecting value
σ_k	standard deviation

Genetic Algorithms

$\mathbf{P}_{...}$	= Parent = a solution proposal
O	offspring issued out of two parents
$\mathbf{S}_{...}$	subtree = character string extracted out of P
G(p)	degree of order

take from S. 23

Neuronal Algorithms

$Z_e(s)$	= area of influence
S_k	layer
&	AND–operation
1$\leq$	OR–operation

Used Abbreviations

NP–complete	non–polynomial–complete
SA	Simulated Annealing
GORDIAN	Global Optimization and Rectangle Dissection
SAP	Somatotopical projection for Placement
HNP	Hierarchical neuronal Placement

List of Tables

List of Figures

Bibliography

[Bril90] F. Z. Brill, D.E. Brown, W. N. Martin : Genetic Algorithms for feature selection for counterpropagation Networks. In *Institue of parallel computation University of Virginia*, pages 90–105, 1990

[Coho91] J. P. Cohoon, S. U. Hedge, W. N. Martin D. S. Richards. Distributed Genetic Algorithms for the Floorplan Design Problem. In *IEEE Transactions on CAD,* pages 483–491, Vol. 10, No. 4, April 1991

[Wong86] D. F. Wong and C. L. Liu. A New Algorithm for Floorplan Design. In *IEEE Transactions on CAD,* pages 483–491, Vol. 10, No. 4, April 1991

[wikiNP] https://en.m.wikipedia.org/wiki/P_versus_NP_problem .

[wikiGA] https://en.m.wikipedia.org/wiki/Genetic_algorithm .