Abstract

Genetic algorithms are stochastic search and optimization techniques which can be used for a wide range of applications. This paper addresses the application of genetic algorithms to the graph partitioning problem. Standard genetic algorithms with large populations suffer from lack of efficiency (quite high execution time). A massively parallel genetic algorithm is proposed, an implementation on a SuperNode® of Transputers® and results of various benchmarks are given.

The parallel algorithm shows a superlinear speed-up, in the sense that when multiplying the number of processors by p, the time spent to reach a solution with a given score, is divided by kp (k>1).

A comparative analysis of our approach with hill-climbing algorithms and simulated annealing is also presented. The experimental measures show that our algorithm gives better results concerning both the quality of the solution and the time needed to reach it.

Key Words

Distributed memory parallel architectures, Genetic algorithms, Graph partitioning, Hill-climbing, Mapping problem, Simulated annealing, Superlinear speed-up.

1. This work has been supported by ESPRIT project SuperNode2 (P2528).
2. Address: BP53X, F-38041 Grenoble.FRANCE; Phone: (33)76.51.45.72.; Email: ghazali@imag.fr & bessiere@imag.fr.

I. INTRODUCTION

Given a graph, the "graph partitioning problem" searches for a partition of its nodes which optimizes a given cost function.

There are numerous practical applications of this problem, for instance:
- design of VLSI (Very Large Scale Integration) circuits, where, given a set of components and a set of modules, one wants to place the components in order to minimize the number of connections between modules, yet preserving some balance concerning the number of components on each module [Lawler69][Russo71];
- routing in distributed systems, where the considered problem is to subdivide the computer network into smaller clusters so that the control overhead for routing is minimized [Bouloutas89];
- image segmentation in the field of computer vision, where segmented images are represented as graphs in which each vertex represents a segment and each weighted edge between two vertices represents a topological relationship between two segments of the image [Héralu89];
- virtual memory paging systems, where one wants to distribute the different objects on memory pages in order to minimize the number of references between objects stored on different pages [MacGregor78];
- mapping parallel programs on parallel architectures.

In our laboratory, we are especially interested in this last application, namely, the placement of communicating processes on processors of a distributed memory parallel machine. A survey of the different methods proposed in the literature to deal with this problem may be found in [Talbi90]. The parallel program is modeled as a graph where the vertices represent the processes, the vertices' weights represent known or estimated computation costs of these processes, the edges represent communication links required between them and the edges' weights estimate the relative amount of communication necessary along those links. When the number of processes exceeds the number of available processing elements, as it is usually the case in massively parallel programming, the mapping problem
includes the contraction problem [Berman87] which is
equivalent to the graph partitioning problem treated in this
paper.

The graph partitioning problem is NP-complete.

Consequently, heuristic methods should be used to deal
with it. They may find solutions that are only approximations of
the optimum, but they will do it in a reasonable amount of time.
The different approaches that have been proposed for this problem
may be divided in two
main classes. On one hand, the general purpose
optimization algorithms independent of the given application
and, on the other hand, the heuristic approaches
especially designed for a unique problem. As we want to
avoid the intrinsic disadvantage of the algorithms of this
second class (their limited applicability due to the problem
dependence) our concern in this paper, is only the first class
of algorithms.

Two widely used optimization techniques are the hill-
climbing algorithm [Haden88] and simulated annealing
[Sheild87]. Hill-climbing is sure to find the global
minimum only in convex spaces. Otherwise, most often it
is a local rather than a global minimum which is found.
Simulated annealing offers a way to overcome this major
drawback of hill-climbing but the price to pay to do so is
an important computation time. Worst, simulated
annealing algorithm is rather of a sequential nature, its
parallelization is quite a difficult task
[Biaard89][Savage90].

More distributed optimization techniques may also be
considered. Some of them are closely related to neural
networks algorithms (see [Ackley87] & [Peretto90]). Others,
many, genetic algorithms (GAs) are considered in
this paper. They are stochastic search techniques, introduced
by Holland twenty years ago [Holland75], inspired by
biological evolution of species. Development of massively
parallel architectures made them very popular in the very
last years. They have recently been applied to combinatorial
optimization problems in various fields, such as, for instance,
the traveling salesman problem [Grefenstette87],
the optimization of connections and connectivity of neural
networks [Whitley90], and classifier systems
[Robertson87].

The purpose of this paper is to prove that the graph
partitioning problem may be solved quite efficiently by a
parallel genetic algorithm.

The structure of the paper is as follows:
- in a first section, we give a mathematical
formalization of the graph partitioning problem and discuss
few instances of classical cost functions.
- in the next section, we extensively present the
  genetic algorithm approach to the graph partitioning
  problem. After a recall of the genetic algorithms principles,
  we show how they can be used to deal with the graph
  partitioning problem, we discuss the matter of parallel
  genetic algorithms and finally we expose the proposed
  solution.
- the third and final section, after a detailed
  presentation of the Supernode implementation, presents the

results of different benchmarks comparing either speed or
quality of results of this algorithm given different sizes of
population or given different numbers of processors. A
comparative analysis of the genetic algorithm solution with
hill-climbing and simulated annealing ones is also
presented.

Finally, concluding remarks and possible extensions
to this work are proposed.

II. MATHEMATICAL FORMALIZATION OF
THE GRAPH PARTITIONING PROBLEM

Given:
- an undirected graph $G = (V,E)$;
- an application $\Omega_1$ from $V$ into $Z^+$, such that
  $\Omega_1(v_i) = w_{1i}$ is the weight of vertex $v_i$;
- an application $\Omega_2$ from $E$ into $Z^+$, such that
  $\Omega_2(e_j) = w_{2j}$ is the weight of edge $e_j$;
- and a set of numerical constraints $\Phi = \{\phi_1, \phi_2,
  ..., \phi_m\}$ on these weights;

the graph partitioning problem has to find a
partition $\Pi$ of $V$ ($\Pi = \{\pi_1, \pi_2, ..., \pi_n\}$) satisfying the
constraints $\Phi$.

A classical and well studied set $\Phi_1$ of constraints
expresses that:
- for each sub-set $\pi_i$ of $V$ belonging to the partition
  $\Pi$, the sum of the weights of its vertices must be inferior
to a given value $B$

$$\forall \; \pi_i \in \Pi, \; \sum_{v \in \pi_i} \Omega_1(v) \leq B$$

- the sum of the weights of edges going from one
  node of $\pi_1$ to one node of some other $\pi_i$ must be inferior
to some given value $C$

$$\sum_{e \in \pi_i} \Omega_2(e) \leq C$$

with $e = \{ (x,y) / (x,y) \in E \; \& \; x \in \pi_i \; \& \; y \in \pi_i \; \& \; x \neq y \}$

The graph partitioning problem under constraints $\Phi_1$
has been proved NP-complete (see [Garey79]).

Most applications correspond to the following set $\Phi_2$
of constraints where the weights of all nodes are set to 1
(see [Kernighan70] & [Feo90]):
- for each sub-set $\pi_i$ of $V$ belonging to the partition
  $\Pi$, the number of nodes in $\pi_i$ is equal to a given value $B_i$

$$\forall \; \pi_i \in \Pi, \; \sum_{v \in \pi_i} \Omega_1(v) = B_i$$

with $\forall \; v \in V, \; \Omega_2(v)=1$

- the total cost of the edges going from one $\pi_i$ to
  another $\pi_j$ should be minimum

$$\text{MIN} \left( \sum_{e \in \pi_i} \Omega_2(e) \right)$$

The graph partitioning problem under constraints $\Phi_2$
has also been proved NP-complete (see [Hyafil73]).

For our application, the mapping of parallel programs
on parallel architectures, we have to consider the following
set $\Phi_3$ of constraints:
- minimize the sum of the total communication costs between processors (total cost of the edges going from one \( \pi_i \) to another \( \pi_j \)) and of the variance of the loads of the different processors (variance of cost of vertices belonging to a given \( \pi_i \)):

\[
\min\left(\sum_{\pi_i \in \Pi} \Omega_{\pi_i} + (K \cdot \left( \frac{\sum_{\pi_i \in \Pi} \Omega_{\pi_i}}{|\Pi|} \right)^2 \cdot \left( \frac{\sum \Omega(v)}{|\Pi|} \right)^2) \right)
\]

With \( K=0 \) the set of constraints \( \Phi_3 \) reduces to \( \Phi_2 \). This proves that the partitioning problem under constraints \( \Phi_3 \) is NP-complete.

For the mapping problem, \( K \) is the weight of the contribution of the communication cost relative to the computational load balance across the system. Choosing a suitable value for \( K \) depends on knowledge about characteristics of the parallel architecture. Very small values of \( K \) would suggest a uniprocessor solution, and very large values would reduce the problem to one of multiprocessor scheduling without communication costs. The parallel architecture used was a network of transputers and \( K=2 \) has been chosen.

III. GENETIC ALGORITHM SOLUTION TO THE GRAPH PARTITIONING PROBLEM

III.1. Genetic algorithms principles and their application to the graph partitioning problem

Genetic algorithms compose a very interesting family of optimization algorithms. Their basic principle is quite simple.

Given a search space \( \Sigma \) of size \( M^N \), given \( M \) symbols, any point of this space may be represented by a vector of \( N \) of these \( M \) symbols.

Given a fitness function \( F \) from \( \Sigma \) into \( \Re \) associating a real value to any point of \( \Sigma \).

Given an initial set of vectors, called the initial population.

Some genetic operators are used to generate new points of \( \Sigma \) given some old ones in a phase of the process called "reproduction". During this phase, some points of \( \Sigma \) are replaced keeping the size of the population fixed. The fundamental principle of GA is: "the fitter a vector, the most probable its reproduction". In mathematical terms it means that the probability \( P \) of reproduction is increasing as \( F \) increased:

\[
\forall \sigma_1, \sigma_2 \in \Sigma, \quad F(\sigma_1) > F(\sigma_2) \Rightarrow P(\sigma_1) > P(\sigma_2)
\]

The standard genetic algorithm is:

Generate a population of random individuals.
While \( \text{number of generations} \leq \text{max number of generations} \)
Do

- **Evaluation** - assign a fitness value to each individual.
- **Selection** - make a list of pairs of individuals likely to mate, with fitter individuals listed more frequently.
- **Reproduction** - apply genetic operators to the selected pairs.
- **Replacement** - form a new population by replacing worst individuals by best ones.

The most common genetic operators used during reproduction are crossover and mutation. Crossover, given two vectors, cut them both at the same random point and exchange the two portions thus cut out (fig.1a). Mutation is simply flipping a bit (fig.1b). Two parameters need to be defined: \( P_c \) and \( P_m \). They represent respectively the probability of application of the crossover and mutations operators. Other genetic operators may be find in the literature, for instance, the inversion operator [Cohon87] and many variants of the crossover operator designed for specific problem domains [Shahook90].

![Crossover and Mutation Operators](image)

(a) Crossover

(b) Mutation

Figure 1: Genetic operators.

To use genetic algorithm for the graph partitioning problem, the following formalism is used:

Let us suppose that we have a graph of \( N \) nodes to divide into \( M \) sub-sets. Each of these sub-sets is labelled by a symbol (for instance an integer between 0 and \( M-1 \)). A given partition is represented by a \( N \) vector of these symbols; where symbol \( p \) in position \( q \) means that node \( q \) of the graph is in sub-set \( p \).

The fitness function \( F \) is the last cost function described in section II. We use the usual version of crossover, but
mutation is a random trial of one of the M possible symbols.

III.2. A parallel genetic algorithm

Two approaches to parallel genetic algorithms have been considered so far:

**Standard Parallel Approach:** In this approach, the evaluation and the reproduction are done in parallel. However, the selection is still done sequentially, because selection would require a fully connected graph of neighbors as any two individuals in the population may be mated.

**Decomposition Approach:** This approach consists in dividing the population into equal size subpopulations. Each processor runs the genetic algorithm on its own sub-population, periodically selecting good individuals to send to its neighbors and periodically receiving copies of its neighbors' good individuals to replace bad ones in its own sub-population. The processor neighborhood, the frequency of exchange, and the number of individuals exchanged are adjustable parameters.

The standard parallel model is not flexible in the sense that the communication overhead grows as the square of population's size. Therefore, this approach is not adapted to distributed memory architectures, where the cost of communication has a great impact on the performance of parallel programs. In the decomposition model, the inherent parallelism is not fully exploited as treatment of subpopulations may be further decomposed. This approach should be considered only when the number of available processors is less than the required size of the population.

Considering massively parallel architectures with numerous processors, we chose a coarse-grained model, where the population is mapped on a connected processor graph like a grid, one individual per processor. We have a bijection between the individual set and the processor set. The selection is done locally in a neighborhood of each individual. Another version to this approach has been already proposed in [Mühlenbein89], where at each generation a hill-climbing algorithm is executed for each individual in the population.

The choice of the neighborhood is the adjustable parameter. To avoid overhead and complexity of routing algorithms in parallel distributed machines, a good choice may be to restrict neighborhood to only directly connected individuals.

The parallel genetic algorithm proposed is:

Generate **in parallel** a population of random individuals. While **nber_of_generations** ≤ **max_nber_of_generations**

Do

**Evaluation** - Evaluate **in parallel** each individual.

**Selection** - Receive **in parallel** the individuals coming from its neighbors.

**Reproduction** - Each individual reproduces **in parallel** with the individuals previously received.

**Replacement** - Do **in parallel** a selection of best local offsprings.

It is important to notice that these modifications of the standard model do not cause a degradation in the search efficiency of the standard genetic algorithm as shown in [Anderson90] and [Mühlenbein88].

IV. SuperNode Implementation and Benchmarking

IV.1. SuperNode Implementation

The SuperNode is a loosely coupled, highly parallel machine based on transputers (fig.2). One of its most important characteristics is its ability to dynamically reconfigure the network topology by using a programmable VLSI switch device. This architecture offers a range of 16 to 1024 processors, delivering from 24 to 1500 Mflops performance. To achieve these performance, a hierarchical structure has been adopted [Jesshope86]. The basic component is a T800 transputer. It is a 32-bit microprocessor, with on-chip memory and F.P.U. (Floating Point Unit), delivering 10Mips and 1.5Mflops peak performance. Communication between transputers is supported by 4 bidirectional, serial, asynchronous, point-to-point connection links. A Sun host station is used to provide the connection between the root processor and the external world.

![Figure 2: SuperNode Parallel Architecture.](image-url)
The programming environment used in our experiments is Parallel C 3L. A configurator of the physical network, developed in our laboratory has been used to obtain the desired topology of the architecture.

We assume that each individual in the population resides on a processor and communication is carried out by message passing. The following is a pseudo-OcCam description of the process executed in parallel by each processor:

SEQ
  Generate (local_individual)
  Evaluate (local_individual)
  While nber_of_generations ≤ max_nber_of_generations
    SEQ
      -- communication phase
      PAR i=0 FOR nber_of_neighbors
        PAR
          neighbor_in[i] ? neighbor_individual[i]
          neighbor_out[i] ! local_individual
      -- computation phase
      PAR i=0 FOR nber_of_neighbors
        Reproduction(local_individual,
        neighbor_individual[i])
      Replacement

    Each reproduction produces two offsprings. Our strategy is to choose randomly one of the offsprings. The replacement phase consists in replacing the current local individual with the best local offspring produced in the reproduction phase.

    The population is placed on a torus. Given the four links of the transputer, each individual has four neighbors. No routing is needed in the processor network because only directly connected processors have to exchange information.

    We do not consider the best solution found globally since the communication involved in determining this solution would be considerable. We only pick up the best solution routing through a "spy process" placed on the "root processor" (see fig. 3).

    Figure 3: A torus of 16 processors.

IV.2. Varying the number of processors

The purpose of this benchmark is to measure the speed-up when running the genetic parallel algorithm (for a given population size) on different sizes of torus of processors.

We use the speed-up ratio as a metric for the performance of the parallel genetic algorithm. The speed-up ratio S is defined as S=Ts/Tp where Ts is the execution time on a single processor and Tp corresponds to execution time for p processors implementation. Figure 4 shows the obtained results.

The algorithm has a near-linear speed-up. This is due to the fact that the communication cost between processes is relatively small compared with the computation cost, and is independent of the size of the architecture.

Figure 4: Speed-up of the parallel algorithm

IV.3. Varying the population size

The purpose of this benchmark is to measure the evolution of solution's quality when running the parallel genetic algorithm with different sizes of population.

Figure 5 shows the obtained results.

Solution quality

Figure 5: Solution quality function of population size.
Notice that given the specific graph partitioning problem used (a pipeline of 32 vertices to be partitioned in 8 sub-sets) for this benchmark, the best possible solution scores 7.

As expected, for a given number of generations, the quality improves with an increase of population's size.

It may even happen that for a too small population a premature convergence occurs and that the optimal solution will not be ever reached.

The figure 5 shows also that the greatest reduction in the cost of the partitioning occurs at the beginning. Thus a moderate quality partitioning can be obtained very quickly.

IV.4. Time to reach a given solution

The purpose of this benchmark is to study the speed-up for a given solution's quality when running the genetic parallel algorithm on different sizes of torus of processors and with different sizes of populations (both being equal given that there is one individual per processor).

Figure 6 shows the influence of the number of processors (and population size) on the time needed to reach a solution scoring 8.

<table>
<thead>
<tr>
<th>number of processors</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>304</td>
</tr>
<tr>
<td>16</td>
<td>164</td>
</tr>
<tr>
<td>32</td>
<td>71</td>
</tr>
<tr>
<td>64</td>
<td>33</td>
</tr>
</tbody>
</table>

IV.5. Comparison with hill-climbing and simulated annealing algorithms

In this section, the performance of the proposed algorithm is compared with hill-climbing and simulated annealing algorithms.

a/ Hill-climbing algorithm

The hill-climbing algorithm starts with a random configuration, and tries to improve it [Johnson85]. The improvement is carried out in small steps consisting of moving a vertex from one sub-set to another. A move is selected randomly, the cost change of the move is evaluated, and if the change is for the better move is accepted and a new configuration is generated. Otherwise, the old configuration is kept. This process is repeated until there are no changes to the configuration that will reduce the cost function further. When this occurs a local minimum has usually been found, rather than the required global minimum. Figure 7 shows a version of the hill-climbing algorithm.

Generate a random initial state $S_0$ ($S_i=S_0$).

Repeat

- compute at random a neighboring state $S'$.
- if $\text{cost}(S') < \text{cost}(S)$ then $S=S'$

Until there is no better neighbor.

Figure 7: The hill-climbing algorithm.

b/ Simulated annealing algorithm

The principle of the simulated annealing algorithm is the following [Kirkpatrick83] (fig.8): the system is in a high temperature environment. At this temperature is applied a sufficiently long sequence of random elementary transformations (Markov chain) to reach the equilibrium. At this temperature, the system is slightly increased and a new sequence of random move is applied. At each temperature the permitted energy states are governed by the Metropolis criterion, which allows the configuration to accept a state with a probability $P(\Delta E,T)$. The search terminates when the system stabilizes.

There is a large amount of literature on this topic [Laarhoven87], and the basic algorithm allows for considerable variation and tuning of parameters. The version we considered is a straightforward version, which can probably be greatly improved. However, as the parallel genetic algorithm is also a "naive", "out-of-the-shell" version, we think that the following benchmarks give, at least, interesting order of magnitudes.

The number of available changes to the configuration, denoted by $L$, when moving one vertex to a different subset, is given by $L=N(M-1)$, where $N$ is the number of vertices of the graph to be partitioned and $M$ the number of sub-sets of the partition. This value gives a measure of the size of the problem and is used as a parameter in the annealing schedule.

Figure 6: Execution times on different sizes of parallel architectures.

We have a "superlinear" speed-up of the parallel genetic algorithm, in the sense that when multiplying the number of processors by $p$ the execution time is divided by $kp$ ($k>1$).
1. (Initialization step)
   - start with a random initial configuration S0 (S=S0);
   - $T = T_{max}$;
2. (Stochastic hill-climb)
   - generate and compute a random neighboring state S';
     $\Delta E = \text{cost}(S') - \text{cost}(S)$
   - select the new configuration (S=S') with probability
     $P(\Delta E, T) = \min(1, \exp(\Delta E/T))$;
   - repeat this step $X \times N \times (M-1)$ times; /* length of the
     markov chain spent at each T */
3. (Anneal/Convergence test)
   - set $T = T_{min}$;
   - if $T \geq T_{min}$ goto step 2.

Figure 8 The simulated annealing algorithm.

**c/ Experimental protocol and results**

Each algorithm was run 10 times to obtain an average performance estimate. Experiments were performed on two different problems:

- a pipeline of 32 vertices to be partitioned in 8 sub-sets;
- a grid of 64 vertices to be partitioned in 4 sub-sets.

For the genetic algorithm, we use a population of 64 configurations running on a 8 by 8 torus of processors. The annealing schedule and the genetic algorithm parameter's that have been used during our experiments are given by table 1 and 2.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{max}$</td>
<td>10</td>
<td>Starting temperature</td>
</tr>
<tr>
<td>$T_{min}$</td>
<td>0.1</td>
<td>Minimum temperature</td>
</tr>
<tr>
<td>$a$</td>
<td>0.9</td>
<td>Temperature decay rate</td>
</tr>
<tr>
<td>$X$</td>
<td>2</td>
<td>Length of the markov chain</td>
</tr>
</tbody>
</table>

Table 1: The annealing schedule.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP8</td>
<td>64</td>
<td>Population size</td>
</tr>
<tr>
<td>$P_c$</td>
<td>1</td>
<td>Crossover probability</td>
</tr>
<tr>
<td>$P_m$</td>
<td>0.5</td>
<td>Mutation probability</td>
</tr>
</tbody>
</table>

Table 2: Genetic algorithm parameter's.

The tables below show the minimum, maximum, average value and the variance of the obtained solutions. The results for the hill-climbing and the simulated annealing algorithms are based on an implementation on a single T8000 transputer.

It can be observed from tables 3 and 4 that the genetic algorithm outperforms hill-climbing and simulated annealing algorithms both in the quality of the solution and the time used in the search.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>min</th>
<th>max</th>
<th>mean</th>
<th>deviation</th>
<th>CPU time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hill-climbing</td>
<td>7.5</td>
<td>12.5</td>
<td>9.9</td>
<td>1.94</td>
<td>353</td>
</tr>
<tr>
<td>Simulated annealing</td>
<td>7.5</td>
<td>9</td>
<td>8.05</td>
<td>0.37</td>
<td>2296</td>
</tr>
<tr>
<td>Genetic algorithm</td>
<td>7</td>
<td>8</td>
<td>7.5</td>
<td>0.15</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 3: Benchmarking with a pipeline of 32 vertices and a partition of 8 sub-sets.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>min</th>
<th>max</th>
<th>mean</th>
<th>deviation</th>
<th>CPU time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hill-climbing</td>
<td>24</td>
<td>47</td>
<td>32.9</td>
<td>42.49</td>
<td>643</td>
</tr>
<tr>
<td>Simulated annealing</td>
<td>21</td>
<td>27</td>
<td>24</td>
<td>4.00</td>
<td>3358</td>
</tr>
<tr>
<td>Genetic algorithm</td>
<td>17</td>
<td>25</td>
<td>21</td>
<td>6.25</td>
<td>93</td>
</tr>
</tbody>
</table>

Table 4: Benchmarking with a grid of 64 vertices and a partition of 4 sub-sets.

For simulated annealing, best results may be obtained using bigger values of X, however, raising X will increase the computation time.

V. CONCLUSIONS AND FUTURE DIRECTIONS

A parallel genetic algorithm to solve the graph partitioning problem has been proposed and evaluated.

The main results are the following:
- the algorithm shows a superlinear speed-up;
- it is easy to program;
- it is simple to implement on massively parallel distributed memory architectures;
- it outperforms hill-climbing and simulated annealing algorithms both in the quality of the solution and the time used to reach it.

An important characteristic of genetic algorithms is that they may be used to solve a great variety of combinatorial optimization problems. We are using them to solve such optimization problems in the field of robot control and neural networks [Bessière90].

We are also studying an important improvement of the algorithm, namely, the dynamic variation of its parameters and particularly mutation probability. The crossover operator becomes less effective over time as the strings in the population become more similar. One way to avoid the premature convergence and to sustain genetic diversity is by using adaptive mutation. During the first generations when there is ample diversity in the population, mutation must occur at very low rates. However, as diversity decreases in the population, the mutation rate must increase.

More theoretical work is planned: a cellular automata based model will be used to study the influence of the algorithm's parameters on its convergence.
BIBLIOGRAPHY


