Selçuk J. Appl. Math.
Vol. 3, No. 2, pp. 3–22, 2002

Selçuk Journal of Applied Mathematics

Efficient strategies for optimization with genetic algorithms

Hans-Joachim Bungartz and Igor Trajkovski

Institute of Parallel and Distributed Systems, University of Stuttgart,
Breitwiesenstr. 20–22, D 70565 Stuttgart, Germany;
e-mail: bungartz@informatik.uni-stuttgart.de;
e-mail: trajkoir@informatik.uni-stuttgart.de

Received: November 9, 2002

Summary. Evolutionary strategies in general and genetic algorithms in particular have turned out to be of increasing relevance for various classes of optimization problems like combinatory problems as a discrete example or shape optimization as a continuous example. In this paper, we present efficient and powerful strategies for genetic algorithms and their application to two classes of optimization problems. Besides algorithmic aspects concerning the genetic essentials, the focus is put on the efficient implementation, both of the sequential and of the parallel versions.

Key words: optimization, genetic algorithms, parallel algorithms

2000 Mathematics Subject Classification: 68W10, 90C27, 68R05

1. Introduction

There is a large class of interesting problems for which no reasonably fast algorithms have been developed so far. Many of these problems are optimization problems that arise frequently in applications. Given such a hard optimization problem it is, however, often possible to find an efficient algorithm whose solution is at least approximately optimal. In some cases, we can use probabilistic algorithms as well — these algorithms do not guarantee the optimum value, but by randomly choosing a sufficient number of "witnesses", the probability of error can be made arbitrarily small. There are a lot of important practical optimization problems for which such algorithms of high quality have become available. For instance, we can apply simulated annealing for wire routing and component placement problems in VLSI design or for the traveling salesman problem. Moreover, many other large-scale combinatorial optimization problems (many of which have been proved to be NP-hard) can be solved approximately on present-day computers by such kind of Monte Carlo techniques.

In general, any problem to be solved can be considered as a search through a space of potential solutions. Since we are after the best solution in some sense, we can regard this task as an optimization process. For small spaces, classical enumeration techniques usually suffice; for larger spaces, special artificial intelligence techniques must be applied. Genetic Algorithms (GA) are among such methods: They are stochastic algorithms whose search strategies model some natural phenomena like genetic inheritance or Darwinian strife for survival. Genetic algorithms use a vocabulary borrowed from natural genetics. We speak of individuals in a population; quite often these individuals are called also strings or chromosomes. Chromosomes are made of units — genes (also features, characters) — arranged in linear order. Each chromosome represents a potential solution to a problem; an evolution process run on a population of chromosomes corresponds to a search through a space of potential solutions.

Parallelism, one of the most important paradigms of computer science, has a big potential in GA. Genetic algorithms are very suitable for parallel implementations, as Goldberg [4] observed: "In a world where serial algorithms are usually made parallel through countless tricks and tips, it is no small irony that genetic algorithms (highly parallel algorithms) are made serial through equally unnatural tricks and tips".

In this paper, we discuss both the potential of state-of-the-art strategies for optimization with genetic algorithms and their efficient implementation in both the sequential and the parallel case. The remainder of the paper is organized as follows. In Section 2, we provide a brief overview of the most important components and properties of genetic algorithms. Then, Section 3 is dedicated to sequential algorithms and results, with a focus on the Traveling Salesman Problem (TSP) and the Quadratic Assignment Problem (QAP). Finally, Section 4 presents results for parallel strategies in genetic algorithms, followed by some concluding remarks.

2. Genetic algorithms

Different main schools of evolutionary algorithms evolved during the last 30 years: genetic algorithms, mainly developed in the USA by J. H. Holland [7], evolutionary strategies, developed in Germany by I. Rechenberg [14] and H.-P. Schwefel [16], and evolutionary programming [2]. Each of these constitutes a different approach, however, they are inspired by the same principles of natural evolution. A good introductory survey can be found in [1].

Evolutionary algorithms are stochastic search methods that mimic the metaphor of natural biological evolution. Evolutionary algorithms operate on a population of potential candidates (points) applying the principle of survival of the fittest to produce better and better approximations to the given problems. At each generation, a new set of approximations is created by the process of selecting individuals according to their level of fitness in the problem domain and breeding them together using operators borrowed from natural genetics. This process leads to the evolution of populations of individuals that are better suited to their environment than the individuals that they were created from, just as in natural adaptation. Evolutionary algorithms model natural processes, such as selection, recombination, mutation, migration, locality, and neighbourhood. Figure 1 shows the structure of a simple genetic algorithm. Evolutionary algorithms work on populations of individuals instead of on single solutions. In this way, the search is defined in a parallel manner. At the beginning of

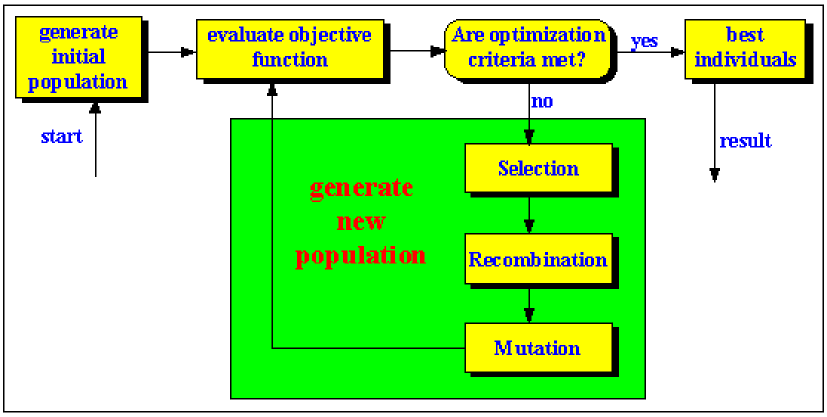


Fig. 1. Structure of a single-population evolutionary algorithm

the computation, a number of individuals (the population) are randomly initialised. The objective function is then evaluated for these individuals, the first or initial generation is produced.

If the optimality criteria are not met, the creation of a new generation starts. Individuals are selected for the production of offspring according to their fitness. Parents are recombined to produce offspring. All offspring will be mutated with a certain probability. The fitness of the offspring is then computed. The offspring are inserted into the population replacing the parents, thus leading to a new generation. This cycle is performed until the optimality criteria or a maximum number of iterations are reached.

Such a single-population evolutionary algorithm is powerful and performs well on a broad class of problems. However, experiments show that better results can be obtained by introducing many populations, called subpopulations. Every subpopulation evolves for a few generations in an isolated way (like in the single-population evolutionary algorithm) before one or more individuals are exchanged between the subpopulations. Typically such a multipopulation evolutionary algorithm models the evolution of a species in a way more similar to nature than the single-population evolutionary algorithm.

2.1. Principles, methods, and algorithms

From the above discussion, it can be seen that evolutionary algorithms differ substantially from more traditional search and optimization methods. The most significant differences are:

- Evolutionary algorithms search a population of points in parallel, not a single point.
- Evolutionary algorithms do not require any kind of analytical regularity or other auxiliary knowledge; only the objective function and corresponding fitness levels influence the directions of search.
- Evolutionary algorithms use probabilistic transition rules, not deterministic ones.
- In general, evolutionary algorithms are more straightforward to apply.
- Evolutionary algorithms can provide a number of potential solutions to a given problem. The final choice is left to the user.

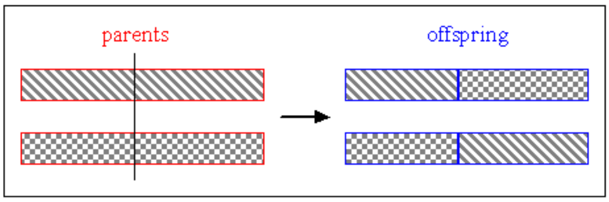
2.1.1. Selection Selection determines which individuals are chosen for mating (recombination) and how many offspring each selected individual produces. The first step of a selection process is fitness assignment, which can be done either proportional to the objective value of the individual [4] or proportional to the individual's rank. In rank-based fitness assignment, the population is sorted according to the objective values. The fitness assigned to each individual depends only on its position in the sorted list and not on the actual objective value.

The actual selection is performed in the next step. For the selection of parents due to their assigned fitness, several strategies have been developed. In *roulette-wheel selection*, the individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness. A random number is generated, and the individual whose segment contains the random number is selected. In *tournament selection*, a subgroup of individuals is randomly selected, and the best individual from this subgroup is selected to be a parent. *Truncation selection* is based on sorting the individuals according to their fitness. Only the best individuals may be selected as parents. The parameter for truncation selection is the truncation threshold ϵ_{trunc} that indicates the proportion of the population to be selected as parents. The usual value of this parameter ranges from 10% to 50%. Individuals below the truncation threshold do not produce offspring. Finally, the so-called *local selection* uses some notion of bounded local neighbourhoods, where all individuals are supposed to live in. A neighbourhood can be seen as a group of potential mating partners. First, half of the parents are selected at random or by using one of the above-mentioned selection mechanisms. Then, a local neighbourhood is defined for every selected individual, inside which the mating partner is selected in an appropriate way.

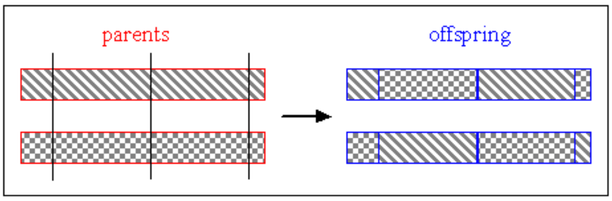
2.1.2. Recombination Recombination produces new individuals in combining the information contained in the parents. Depending on the representation of the variables of the individuals, the following algorithms can be applied:

- Real-valued recombination: *discrete* (performs an exchange of variable values between the individuals, can be used with any kind of variables (binary, real, or symbols)), *intermediate* (offspring $=$ parent$_1 + \alpha \cdot ($parent$_2 -$ parent$_1)$; α is a scaling factor chosen uniformly at random over an interval $[-d, 1+d]$; a good choice is $d = 0.25$; each variable in the offspring is the result of combining the variables according to the above expression with a new α chosen for each variable), and *line* recombination which is similar to intermediate recombination, except that only one value of α for all variables is used.

– Binary- or integer-valued recombination:
  There are three types: *single-*, *multi-point*, and *uniform* crossover.



(a) Single-point crossover



(b) Multi-point crossover

  Single- and multi-point crossover define cross points as places where an individual can be split. Uniform crossover generalizes this scheme to make every place a potential crossover point.

2.1.3. Mutation After recombination, every offspring undergoes mutation. Offspring variables are mutated by small perturbations (size of the mutation step), typically with a rather low probability. The representation of the variables determines the used algorithm. For binary-valued individuals, mutation means the flipping of some of its values. For every individual, the selection of variables to change is made uniformly at random. For real-valued variables, the mutation operator, which is again applied to some of the values only, is defined as follows:

$$\text{mutated variable} = \text{variable} \pm \text{range} \cdot \delta\,.$$

Here, the sign is chosen at random, range denotes half of the size of the variable's domain, and δ is defined as $\delta = \sum_{i=1}^{m} a_i \cdot 2^{-i}$, where $a_i = 1$ with probability $1/m$, else $a_i = 0; m = 20$.

2.1.4. Reinsertion Once the offspring have been produced by selection, recombination, and mutation of individuals from the old population, the fitness of the offspring may be determined. If less offspring are produced than the size of the original population, then in order to maintain the size of the original population, the offspring have to be reinserted into the old population. Similarly, if not all offspring are to be used at each generation or if more offspring are generated than the size of the old population, then a reinsertion scheme must be used to determine which individuals are to exist in the new population.

2.2. Performance and applications

Genetic algorithms show acceptable performance at acceptable costs on a wide range of problems. They are easy to parallelize and superior to other techniques on complex problems with lots of data and parameters, without any presumptions w. r. t. the problem space, with complex relationships between parameters, or with many (local) optima. They can easily incorporate other methods, they can be run interactively (i. e., integrate user-proposed solutions), and most important, provide many alternative solutions. The drawbacks to mention are their lack of a guarantee for finding the optimal solution, their typically weak theoretical basis, their need for parameter tuning, and, without a deeper understanding of the problem, their sometimes high computational cost.

Concerning possible fields of application of genetic or, in general, evolutionary algorithms, these fields can be classified in several main categories: *numerical and combinatorial optimizations*, *planning* (routing, scheduling, packing), *engineering design* (filters, neural networks, structure design), *simulation and identification*, *control* (off-line, on-line), *data mining and classification*, *artificial life*.

These categories are by no means meant to be absolute or definite. They all overlap to some extent, and many applications could rightly appear in more than one category.

3. Sequential algorithms and results

In this chapter, we present several examples of genetic algorithms tailored to specific applications of TSP and QAP. For all of them, we describe the underlying algorithm, we incorporate some problem domain knowledge into them, and we, furthermore, implement these schemes.

3.1. The Traveling Salesman Problem

3.1.1. Problem description The traveling salesman problem (TSP) is a well-known NP-hard combinatorial optimization problem in which a salesman tries to find the shortest closed tour to visit a set of N cities under the condition that each city is visited exactly once. Since the computing time for finding the optimal solution grows exponentially with N, many heuristic approaches have been proposed in order to compute nearby optimal solutions in reasonable time. In the following, we present means of incorporating TSP domain knowledge into a GA and propose an approach in which local search (local hill-climbing) techniques and GAs are combined. The basic idea of this approach is to let the GA perform its search in the space of local minima rather than in the search space of all feasible tours, and to incorporate several heuristics from conventional approaches in solving TSP. For example, instead of generating random tours that form the individuals of the initial population of a GA, it is possible to use a tour construction heuristics, such as nearest neighbour or various insertion heuristics. Furthermore, genetic recombination offers a large potential for using heuristic elements. For example, instead of determining new edges for producing a valid offspring in the crossover operator on a purely random basis, it might be desirable to prefer short edges. It is also possible to replace mutation by one of the well-known tour improvement heuristics such as 2-opt, 3-opt, or the Lin-Kernigan (LK) heuristics [12]. Finally, tour improvement heuristics can also be used for postprocessing: the resulting population or just the best individual may be the target of an additional run of the tour improvement algorithm.

3.1.2. Implementation The following template for constructing such a heuristic GA shows the possibilities of incorporating TSP heuristics into GA:

1. create the initial population by a tour construction heuristics;
2. apply a tour improvement heuristics to the initial population;
3. selection: select parents for mating;
4. recombination: perform a heuristic crossover;
5. mutation: mutate individuals with a given probability;
6. apply a tour improvement heuristics to offspring;
7. replacement: replace some parents with new offspring;
8. if not finished go to step 3;
9. perform some postprocessing step by applying a tour improvement heuristics;

In our experiments we use nearest neighbours as the construction heuristics and LK as the improvement heuristics. Mutation is performed in such a way that, first, X edges are selected randomly (X is a variable with negatively exponential distribution) and that, then, the resulting disconnected parts are randomly reconnected. Furthermore, we implemented a truncation-based replacement strategy, or we produce a certain amount of offspring, say 20%, of the size of the population, replacing the population's weakest individuals.

If the two parents are too similar, the GA may fail to produce a qualitatively new offspring, since after local improvement the offspring may become identical to one of the parents. The probability of getting an identical offspring increases for more effective local search methods like 3-opt and Lin Kernigan. Therefore, one of the disadvantages of this algorithm may be a very fast convergence of the population (all individuals are the same) if we don't use an appropriate recombination operator that will produce offspring which are very different from their parents but, nevertheless, carry some inherited material. Such a recombination operator is DPX (distance preserving crossover), which tries to generate offspring that has equal distance to both of its parents. It works as follows: The content of the first parent is copied to the offspring and all edges that are not in common with the other parent are deleted. The resulting parts of the broken tour are reconnected without using the non-shared edges of the parents. A greedy reconnection procedure is used to achieve this: If the edge (i, j) has been destroyed, the nearest available neighbour k of i among the remaining tour fragments is taken, and the edge (i, k) is added to the tour, provided that (i, k) is not contained in the parents. In order to illustrate the DPX operator, let us consider an example (see Fig. 2). First, a city is chosen randomly as the starting

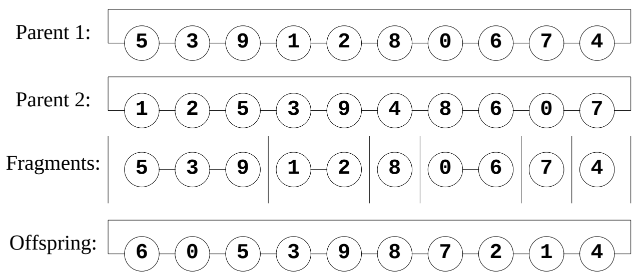


Fig. 2. The DPX crossover

point for the reconnection. Let us assume that the city to begin with is city 6. Then the other endpoint (city 0) of the fragment containing city 6 is considered and its nearest neighbour in the set of available cities, {5,9,1,2,4}, is determined. The set of available cities only contains the start and end points of not yet visited tour fragments. City 8 and city 7 are not contained in this set, because it is not desirable to reinsert edge (0, 8) or edge (0, 7), since they are contained in parent 1 or parent 2, respectively. Let us assume that in the example the nearest neighbour to city 0 is city 5, so city 0 is connected to city 5, and the end of the connected fragment (city 9) is considered. At this point the set of available cities is {2,8,7}. The procedure is repeated until all fragments have been reconnected. Note that the distance d between the offspring and both parent 1 and parent 2 is identical to the distance between the two parents (d = 6), hence the name distance preserving crossover.

3.1.3. Results The proposed approach has been implemented in C programming language on a 2 GHz Pentium 4 under Linux. In Table 1, the results obtained in our experiments are shown. The TSP instances investigated were taken from the TSPLIB [15] (the numbers in the abbreviations indicate the number of the cities). The shortest tours are known for all of them, thus the third and the fourth columns indicate the deviation from the optimum in per cent. For each of the problem instances, 10 independent runs were performed. In the experiments, the population size was set to 30, the number of offspring was chosen to be 20% or 6, and as the mutation rate, we used the value of 30%. The experiments were stopped after 1000 generations, if the optimum solution was not found before. For all the instances,

instance	optimum	best(quality)	average(quality)	t(hh:mm:ss)
lin318	42029	42029 (0.00%)	42029.0 (0.00%)	00:01:35
att532	27686	27686 (0.00%)	27693.7 (0.03%)	00:07:41
rat783	8806	8806 (0.00%)	8807.3 (0.01%)	00:13:34
pcb1173	56892	56893 (0.00%)	56894.7 (0.01%)	00:52:20
fl1400	20127	20127 (0.00%)	20129.5 (0.01%)	01:38:36

Table 1. The results of the presented GA

the optimal solutions have been found with the proposed approach, except for the instance pcb1173. In this case, the best found solution was only slightly longer than the optimum (56893 vs. 56892). In addition to the information shown in Table 1, it is interesting to consider the increase of quality when executing the individual steps of the GA. The length of the best tour generated by the nearest neighbour tour construction heuristics is usually around 20% above the optimum. The subsequent use of the LK improvement heuristics yields solutions that are at the best between 2% and 3% above the optimum. Thus, the GA is responsible for the smallest but hardest part of the optimization process, namely providing the remaining quality improvements to reach the values shown in Table 1.

Although the integration of TSP domain knowledge into GAs seems to be a promising approach, several previous attempts to use TSP heuristics at particular steps of the algorithm were rather discouraging. For example, Grefenstette [6] used a heuristic crossover and got solutions as far as 25% above the optimum for a 50-cities TSP. In [5], he reports results of experiments with a tour construction heuristics for generating the initial population as well as using a heuristic crossover and local hill-climbing as the mutation operator. These results were better than corresponding ones obtained with "blind" genetic search, but still worse than those produced by a simple 2-opt tour improvement heuristics. Suh and van Gucht [17] have applied 2-opt to some individuals of a GA population and reached a quality of 1.73% above the optimum for a 100-cities problem. In [8], Jog, Suh, and van Gucht present better solutions for the same 100-cities problem: Using 2-opt and Or-opt [11] as the tour improvement operator leads to a quality of 1.37% above the optimum on the average. However, the results obtained with these heuristic GA approaches are still rather poor when compared to conventional heuristics. For example, a nearest neighbour heuristics combined with a 2-opt improvement algorithm (implemented as part of our work) applied to the same 100-cities instance as the one mentioned above is able to find a tour with a better quality (0.50% above the optimum) than the average quality of all heuristic GA approaches mentioned.

The probably most effective approach to the TSP is the one proposed by D. S. Johnson [9,10]. His Iterated LK heuristic (ILK) works as follows: A locally optimal solution produced by LK is slightly mutated by a random non-sequential 4-change and LK is applied again to the modified solution. This is repeated until the results are satisfactory. Although ILK has recently been regarded by D. S. Johnson as a "genetic" approach [10], we do not refer to ILK as a GA since the key concepts of GAs, a population of individuals and recombination of (parts of) several individuals, are missing. In ILK, a single individual rather than a population of individuals is considered at each iteration step. Nevertheless, the results obtained by ILK for many TSP instances are truly impressive.

3.2. The Quadratic Assignment Problem

3.2.1. Problem description The Quadratic Assignment Problem (QAP) is a well-known classical combinatorial optimization problem that can be described as follows. We are given a set of n distinct objects $1, 2, ..., n$, which are to be placed uniquely in $m \geq n$ distinct sites. Associated with each pair of objects (i, j) is an interaction (or traffic) intensity T_{ij}. Associated with each pair of sites (S_k, S_l) is a unit traffic cost $D_{S_k S_l}$, often referred to as the distance between S_k and S_l. We may also specify a fixed cost F_{iS_j} associated with the placement of object i in site S_j. If we let A_i denote the site to which object i is assigned in a particular feasible assignment A, the total cost of the assignment A is given by

$$cost(A) = \sum_{i=1}^{n} F_{iA_i} + \sum_{i=1}^{n} \sum_{j=1}^{m} T_{ij} \cdot D_{A_i A_j}.$$

The distance matrix D is not necessarily symmetric; the fixed costs may be zero. There are numerous examples of applications of this problem. If we define the sites to be particular locations on a plane, our objects to be facilities which will interact, the distances to be directed travel costs between locations, and the traffic intensities to be the pairwise amount of travel required between facilities, then we get the well-known facility layout problem. QAP is known to be NP-complete [3]. We describe below an approach for finding consistently good solutions to instances of QAP using a Genetic Algorithm, and we show that this approach yields solutions comparable to the best previously-known heuristics for QAP when applied to test problems from the literature.

3.2.2. Implementation Since we did not use a traditional binary encoding for solution strings, we were able to design reproduction and mutation functions which operate directly on the solution sequence. Mutation and reproduction were completely independent, unlike in most GA implementations where mutation only takes place on a newly formed child. We selected solutions for mutation from the entire population with uniform probability. Mutation took the form of selecting 2X sites at random (X pairs) and reversing the order of all sites within the subsequence determined by the pair of sites. X is a random variable with a negatively exponential distribution. Selection for reproduction was performed with a bias towards choosing the better solutions in the current population. We chose not to select proportional to the objective function's value, as it is done with

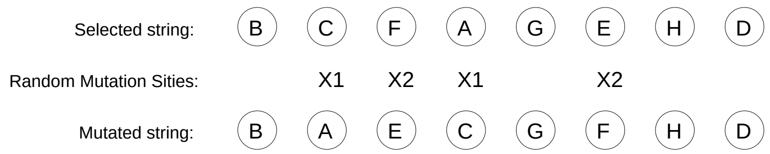


Fig. 3. Mutation

the traditional biased roulette wheel approach. Instead, we selected parent strings by choosing a uniform random number between 1 and $\sqrt{Popsize}$ (where $Popsize$ is the population size), then squaring it. The result was truncated and taken to be the rank of the selected parent (where the first ranked individual is the fittest individual in the population). This approach can be generalized to give an arbitrary degree of preference or even varied dynamically during a given run by changing the power of the root of $Popsize$ during subsequent generations.

We could not adopt the standard crossover since it would not maintain the feasibility of our non-binary coding. Our reproduction scheme was as follows. Any object assigned to the same site in both parents occupies that site in the offspring. For the remaining sites, one of the other objects assigned to that site in one parent is chosen at random, working left to right through the sequence of sites. Any unassigned objects are then matched randomly with the remaining unassigned sites. This reproduction scheme is notable for its ability to generate multiple potentially different children from the same two parents. An example is shown in Fig. 4. For each generation, the

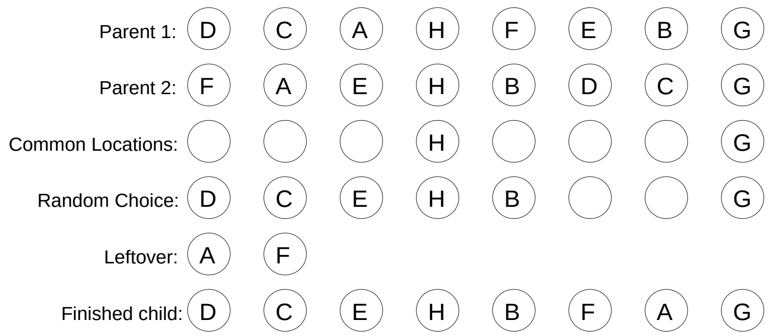


Fig. 4. Recombination

number of children to be produced was selected in advance, as was the probability of a given individual for generating a mutant. A mutant was distinct from the individual which produced it, and both had opportunities to survive. As children and mutants were created, the strings with worst fitness were removed to keep a constant population size.

At the end, as in the previous example TSP, we used a local optimizer applied to every individual. This local optimizer just looks for pairs of objects, which, when we replaced their positions, resulted in a better solution. This procedure is applied until we can not find such a pair which means we are in local minimum defined by our local optimizer.

3.2.3. Results As in the TSP example, the proposed approach has been implemented in C programming language on a 2 GHz Pentium 4 under Linux. In Table 2, the results obtained in our experiments are shown. The QAP instances investigated were taken from the QAPLIB (the numbers in the abbreviations indicate the number of sites). For some of the test data the optimal solution is known, marked with OPT, and for some of them only the upper bounds are known, marked UB. We ran our GA on each problem ten times, because genetic algorithms are stochastic and therefore yield different searches and potentially different results for each random number seed used. These ten runs tested the dependence of the method on the seed selected, both in terms of solution quality and search effort. Each run consisted of a fixed population size of 100 solutions with a randomly generated initial feasible population of 100 individuals. The search was allowed to continue until either the optimum had been reached (if it was known) or 1000 generations had passed.

instance	QAPLIB optimum	best	average	avgerage time (hh:mm:ss)
Bur26a (OPT)	5426670	5426670	5426670	< 00:00:01
Bur26f (UB)	3782044	3782044	3782044	< 00:00:01
Chr25a (OPT)	3796	3796	3848.4	00:00:21
Els19 (OPT)	17212548	17212548	17212548	< 00:00:01
Esc32g (OPT)	6	6	6	< 00:00:01
Esc64a (UB)	116	116	116	00:00:02
Kra32 (OPT)	17740	17740	17740	00:00:23
Lipa50a (OPT)	62093	62093	62093	00:05:43
Nug30 (OPT)	6124	6124	6124.8	00:00:42
Sko56 (UB)	34458	34516	34564.6	00:13:48
Tai30a (UB)	1818146	1818146	1832950.8	00:01:15
Wil100 (UB)	273038	273908	273940.3	01:40:15

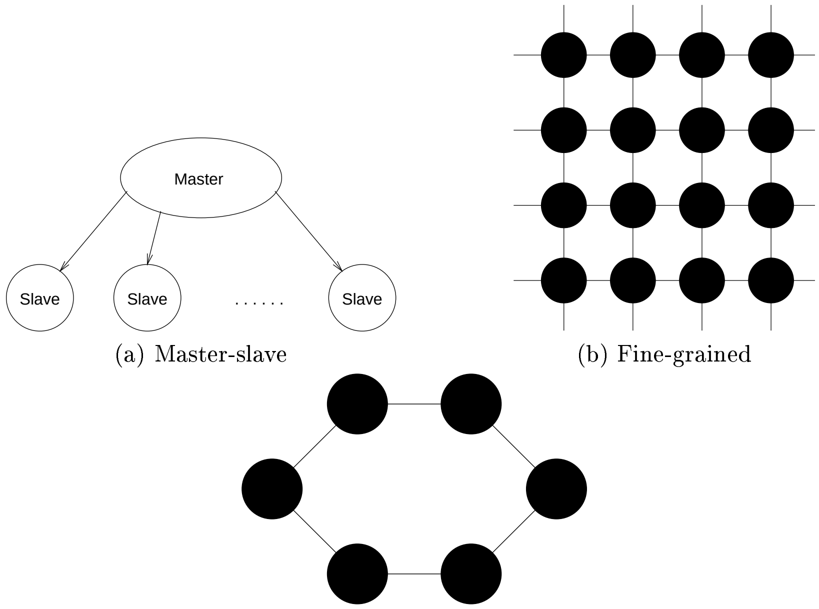
Table 2. The results of the presented GA

The experiments indicate that the presented GA with local search is a promising approach to the QAP. For small instances ($n < 65$), the best-known upper bounds could be found easily, and for large instances the results are quite remarkable. Compared to other published approaches to the QAP, it seems that the proposed genetic algorithm is definitely competitive. In [19], a genetic algorithm is proposed and tested for problems of up to size 30. For the largest problem investigated there (Nug30), the optimum could not be found. In [13], results are reported for solving Nug30 with the Ant System (AS). The best solution found by a combination of AS and simulated annealing had a fitness value of 6128. For large instances, tabu search (TS) has been shown to be the most powerful technique, in particular when optimal solutions are desired. In [18], Taillard compares several variants of tabu search and a hybrid of a genetic algorithm and tabu search.

In summary, we have shown that our genetic approach to TSP and QAP yields solutions comparable to those of the best previously reported heuristics with much lower computational requirements. The best solutions found over a moderate number of test runs were always within a few per cent of the best known solutions, and the average quality of the GA solutions was at least as good as that from previous heuristics for which average case results were reported.

4. Parallel strategies

The basic idea behind most parallel programs is to divide a task into chunks and to solve the chunks simultaneously using several processors. This divide-and-conquer approach can be applied to GA in many different ways, and the literature contains many examples of successful parallel implementations. Some parallelization methods use a single population, while others divide the population into several relatively isolated subpopulations. Some methods can exploit massively parallel computer architectures, while others are better suited to multicomputers with fewer and more powerful processing elements. There are three main types of parallel GA: (1) global single-population master-slave GA, (2) single-population fine-grained, and (3) multiple-population coarse-grained GA (see Fig. 5). In a master-slave GA there is a single population (just as in a simple GA), but the evaluation of fitness is distributed among several processors. Since in this type of parallel GA selection and crossover consider the entire population, it is also known as global parallel GA. Fine-grained parallel GA are suited for massively parallel computers and consist of one spatially structured population. Selection and mating are restricted to a small neighbourhood, but neighbourhoods overlap permitting some interaction among all the individuals. The ideal case is to have only one individual for every processing element available. Multiple-population (or multiple-deme) GA are more sophisticated, as they consist of several subpopulations which exchange individuals occasionally. This exchange of individuals is called migration and it is controlled by several parameters. Multiple-deme GA are very popular, but they are also the class of parallel GA which is most difficult to understand, because the effects of migration are not fully understood. Multiple-deme parallel GA introduce fundamental changes in the operation of the GA and have a different behaviour than simple GA. Multiple-deme parallel GA are known with different names. Sometimes they are known as "distributed" GA, because they are usually implemented on distributed memory MIMD computers. Since the computation-to-communication ratio is usually high, they are occasionally called coarse-grained GA. Finally, multiple-deme GA resemble the "island model" in population genetics which considers relatively isolated demes, so the parallel GA are also known as "island" parallel GA. It is important to emphasize that while the master-slave



(a) Master-slave

(b) Fine-grained

(c) Multiple-population

Fig. 5. Different types of parallel Genetic Algorithms

parallelization method does not affect the behaviour of the algorithm, the last two methods change the way the GA works. For example, in master-slave parallel GA, selection takes into account all the population, but in the other two parallel GA, selection only considers a subset of individuals. Also, in the master-slave case any two individuals in the population can mate (i. e., there is random mating), but in the other methods mating is restricted to a subset of individuals.

Another method to parallelize GA combines multiple population with master-slave or fine-grained GA. We call this class of algorithms hierarchical parallel GA, because at a higher level they are multiple-deme algorithms with single-population parallel GA (either master-slave or fine-grained) at the lower level. A hierarchical parallel genetic algorithm (HPGA) combines the benefits of its components, and it promises better performance than any of them alone. In Fig. 6,

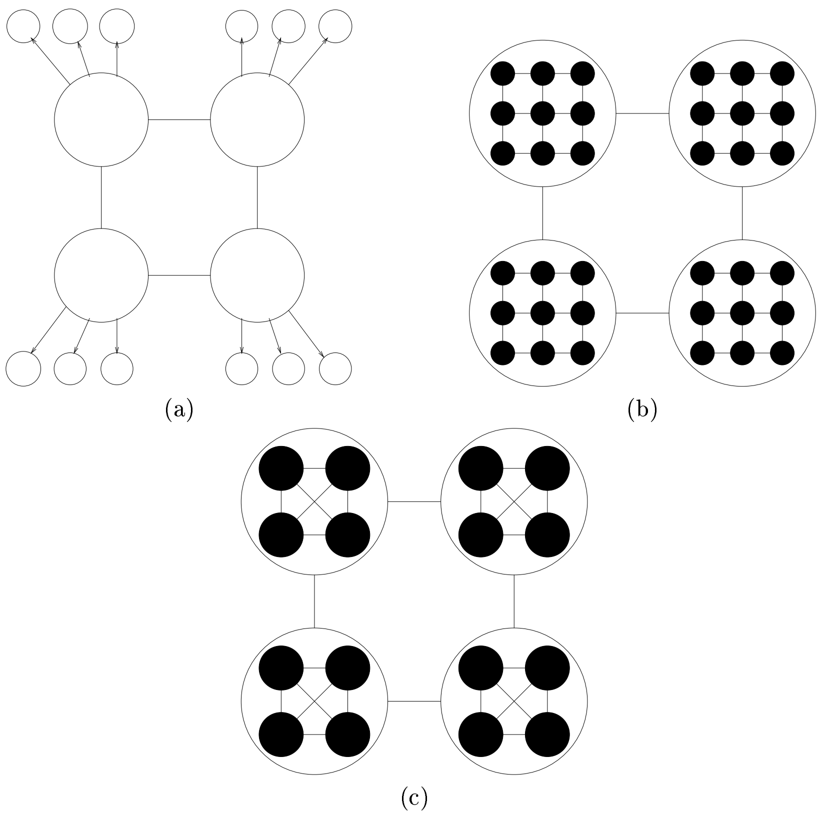


Fig. 6. Hierarchical parallel Genetic Algorithms (HPGA)

HPGA(a) at the upper level is a multi-deme parallel GA where each node is a master-slave GA. HPGA(b) combines a multi-deme GA (at the upper level) and a fine-grained GA (at the lower level). HPGA(c) uses multi-deme GA at both the upper and the lower levels. At the lower level the migration rate is faster and the network topology is much denser than at the upper level.

We implemented a parallel version of our GA using the multiple-population strategy. In our test computations, we ran the same sequential version on several processors, and after every 100 generations 3 randomly selected individuals from 10 best individuals of every population were sent to all other populations. We ran our experiments on 2, 4 and 8 processors, and the results are presented in Table 3, (average time format is (mm:ss)).

instance	#processors		
	2	4	8
Chr25a	00:14	00:09	00:06
Kra32	00:15	00:08	00:04
Lipa50a	03:20	01:45	00:45
Nug30	00:30	00:16	00:11
Sko56	10:02	04:12	02:32
Tai30a	00:40	00:22	00:15
Wil100	55:31	32:47	20:38

Table 3. Parallel results from GA for QAP

The best known solutions are always found for all instances. According to the results, we observe that the algorithm is well fitted to a large number of instances, and that speedup is proportional to the number of the processors. We can mention that there is a case with superlinear speedup (Lipa50a), where the speedup is more than 8 when we run our GA on 8-processor machine. This is due to the fact that GA are non-deterministic algorithms.

5. Concluding remarks

In this paper, we presented efficient strategies for optimization with genetic algorithms, especially for the traveling salesman and the quadratic assignment problem. Concerning their ability to find either the exact optimum or at least a high-quality approximation, they are competitive with the best heuristics known so far. Concerning their computational efficiency, they are clearly superior. For a further increase of performance, we studied several hierarchical parallel GA. Our future work will focus on the application of GA to problems that come along with the numerical treatment of partial differential equations, like mesh generation or load distribution.

References

1. Fogel, D. B. (1994): An introduction to simulated evolutionary optimization, IEEE Trans. on Neural Networks, **5**, No. 1, 3–14.
2. Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966): *Artificial Intelligence through Simulated Evolution*, John Wiley, New York.
3. Garey, M. R. and Johnson, D. S. (1979): *Computers and Intractability. A Guide to the Theory of NP-Completeness.* A Series of Books in the Mathematical Sciences, W. H. Freeman and Company, San Francisco.
4. Goldberg, D. E. (1989): *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading.
5. Grefenstette, J. (1987): Incorporating problem specific knowledge into genetic algorithms, Genetic Algorithms and Simulated Annealing.
6. Grefenstette, J., Gopal, R., Rosmaita, B., and Van Gucht, D. (1988): Genetic algorithms for the traveling salesman problem, in: Genetic Algorithms and Their Applications, Proc. 1st Int. Conf., Pittsburgh, 1985, 160–168.
7. Holland, J. H. (1975): *Adaptation in Natural and Artificial Systems. An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, The University of Michigan Press, Ann Arbor.
8. Jog, P., Suh, J. Y., and Van Gucht, D. (1989): The effects of population size, heuristic crossover and local improvement on a genetic algorithm for the travelling salesman problem, in: Proc. of the 3rd Int. Conf. on Genetic Algorithms, Morgan Kaufmann, 110–115.
9. Johnson, D. S. (1990): Local optimization and the traveling salesman problem, in: Automata, Languages and Programming, Proc. 17th Int. Colloq., Warwick, 1990, Lect. Notes Comput. Sci. **443**, 446–461.
10. Johnson, D. S. and McGeoch, L. A. (1997): The traveling salesman problem: A case study, in: Local Search in Combinatorial Optimization (Aarts, E. et al., Eds.), John Wiley & Sons, Chichester, 215–310.
11. Lawler, E. L. (Ed.), Lenstra, J. K. (Ed.), Rinnooy Kan, A. H. G. (Ed.), and Shmoys, D. B. (Ed.) (1985): *The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization. Reprint*, John Wiley & Sons, Chichester.
12. Lin, S. and Kernigan, B. W. (1973): An effective heuristic algorithm for the traveling salesman problem, Oper. Res., **21**, 498–516.
13. Maniezzo, V., Colorni, A., and Dorigo, M. (1994): *The Ant System Applied to the Quadratic Assignment Problem*, Tech. Rep. 94/28, IRIDIA, University of Brussels.
14. Rechenberg, I. (1973): *Evolutionsstrategie-Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution* [in German], Frommann Holzboog, Stuttgart.
15. Reinelt, G. (1991): TSPLIB — a traveling salesman problem library, ORSA J. Comput., **3**, No. 4, 376–384.
16. Schwefel, H.-P. (1981): *Numerical Optimization of Computer Models*, John Wiley & Sons, Chichester.

17. Suh, J. Y., and Van Gucht, D. (1987): Incorporating heuristic information into genetic search, in: Proc. of the 2nd Int. Conf. on Genetic Algorithms.
18. Taillard, E. D. (1995): Comparison of iterative searches for the quadratic assignment problem, Locat. Sci., **3**, No. 2, 87–105.
19. Tate, D. M. and Smith, A. E. (1995): A genetic approach to the quadratic assignment problem, Comput. Oper. Res., **22**, No. 1, 73–83.