Operator Learning for a Problem Class in a Distributed Peer-to-Peer Environment*

Márk Jelasity[1], Mike Preuß[2], and A. E. Eiben[1]

[1] Free University of Amsterdam, Amsterdam, The Netherlands
jelasity@cs.vu.nl, gusz@cs.vu.nl
[2] University of Dortmund, Dortmund, Germany
mike.preuss@uni-dortmund.de

Abstract. This paper discusses a promising new research direction, the automatic learning of algorithm components for problem classes. We focus on the methodology of this research direction. As an illustration, a mutation operator for a special class of subset sum problem instances is learned. The most important methodological issue is the emphasis on the generalisability of the results. Not only a methodology but also a tool is proposed. This tool is called DRM (distributed resource machine), developed as part of the DREAM project, and is capable of running distributed experiments on the Internet making a huge amount of resources available to the researcher in a robust manner. It is argued that the DRM is ideally suited for algorithm learning.

1 Introduction

This paper discusses a promising new research direction, the automatic learning of algorithm components for problem classes. The main contribution of this paper is threefold. First, we emphasize the importance of an appropriate methodology that allows researchers to produce generalizable knowledge over a problem class rather than a problem instance. Second, we propose a tool that might be ideally suitable for generating such knowledge automatically. Finally, both the methodology and our proposed tool are illustrated via an example: a search operator for a special class of subset sum problem instances is learned.

In the recent years much research effort has been devoted to methods that try to improve heuristic search through some form of learning. To motivate our approach, let us elaborate on its relationship with these methods. The way different approaches generate knowledge can be categorized along at least two dimensions. The first is defined by the distinction between research performed manually and automatic learning. The second is defined by the scope of the knowledge. This scope can be a single problem instance or a whole class of problems. We will call the later type of knowledge *durable* knowledge referring to its generality and long term relevance. As it should be clear already

* The original publication is available at www.springerlink.com. In PPSN VII, LNCS 2439, pp. 172–183, Springer-Verlag, 2002 (doi:10.1007/3-540-45712-7_17). M. Jelasity is also with RGAI, MTA-SZTE, Szeged, Hungary. This work is funded as part of the European Commission Information Society Technologies Programme (Future and Emerging Technologies). The authors have sole responsibility for this work, it does not represent the opinion of the European Community, and the European Community is not responsible for any use that may be made of the data appearing herein.

we will focus on automatically generating durable knowledge here. It is useful however to examine the three other classes generated by the two dimensions described above.

Problem instance specific knowledge generated manually is the unfamous fine tuning process of a given algorithm on a specific problem. Besides of being time consuming, the scientific relevance of such knowledge is questionable due to its restricted scope.

The idea of automatically learning problem instance specific knowledge is very common. It includes all approaches that apply some form of adaptation or learning during the optimization process. The generated knowledge is normally not considered scientifically important and is discarded after the completion of the algorithm. The large field of self-calibrating algorithms (which include the meta-GA approach) belongs to this class [7, 3]. The main idea is that the values of different algorithm parameters are automatically tuned on the fly to achieve optimal performance while solving a problem. Another successful idea is building probabilistic representations of the fitness landscape based on the solutions evaluated during the search and generating new candidate solutions based on this knowledge [14]. If applied as heuristic optimizers, cultural algorithms can be classified into this category as well. They do not fix any knowledge representation offering a more abstract framework for learning based on the performance of the developing population [15].

Durable knowledge is often not generated automatically, partly due to the large computational requirements. One example is [5], where different operators were tested on many random instances of an infinite problem class (NK-landscapes). Using these results it is possible to *predict* the behavior of these operators on *unseen* instances of the problem class. Before optimizing an instance of this class, one can directly chose the best operator.

This paper will argue for the *usefulness* and *feasibility* of *automatically* generating durable knowledge. Usefulness does not need too much explanation: in the case of (practically or theoretically) interesting problem classes this learning can provide us with better algorithms over a whole problem class and can also help us understand this problem class better through the analysis of the collected knowledge. The importance of durable knowledge was emphasized also in [10]. The question of feasibility is not so evident. Producing durable knowledge can be an expensive and slow process. However, distributed systems on wide are networks offer a natural solution to problems that require a huge amount of resources.

With the overall success of the Internet distributed computation is getting more and more attention [6, 17]. Systems exist that can utilize resources available in the form of e.g. the idle time of computers on the Internet [16, 18]. As part of the DREAM project ([13]) such a computational environment was developed, the DRM (distributed resource machine). The DRM—unlike e.g. SETI@home—is based on cutting-edge peer-to-peer (P2P) and Java technology. This allows it to be scalable, robust and flexible [11].

If we consider that many research and engineering institutions solve instances from the same problem class routinely on a large scale anyway, with an additional layer on top of their network (in the form of a distributed computational environment like the DRM) durable knowledge can be generated even more efficiently.

The outline of the paper is as follows. In Section 2 we elaborate on the methodology that allows us to learn durable knowledge. Section 3 is devoted to the DRM, the tool we will use to learn an operator for our example problem class. We also describe our empirical results on the example learning problem. Section 4 concludes the paper.

2 Methodology

In this section we outline a methodology that supports the generation of durable knowledge. We interpret knowledge as algorithmic knowledge, i.e. we want to learn what type of algorithm is optimal on (or at least well-tailored to) a given problem class. More specifically, we are after a good evolutionary algorithm for a problem class, so the search space of our learning task – which we call the *algorithm space* – consists of all EAs. Here we can make further choices and reduce the task to finding good variation operators, recombination and/or mutation. In case of mutation, the algorithm space would be the space of all possible unary operators acting as part of the evolutionary algorithm solving problems of the given class. The size and complexity of the algorithm space depends on how the possible mutation operators are represented. In general, there are no restrictions on this representation. It is possible to define the space simply by a single parameter, e.g. p_m of a fixed bit-flip operator, but using arbitrary expressions from a suitable syntax, e.g. LISP known from genetic programming, is also possible.

Once the algorithm space is defined the learning algorithm cooperates with the basic EA. In general, there are no restrictions on the learning algorithm. In this paper we use an evolutionary algorithm for this purpose, so our method works roughly as follows. A number of problem instances are provided (e.g., by a random instance generator) and different variants of the basic EA is run on these instances. The variants are defined by the mutation operator it uses and these mutation operators form the points of the learning space. The learning EA performs evolutionary search over these points. Evaluating such a point amounts to evaluating the mutation operator it represents by performing independent runs with the given operator on many problem instances. The value of the operator is obtained by the quality of solutions of the basic EA using it. Clearly, this implies very extensive computations which motivates our usage of DRM, cf. section 3.

2.1 An Example Problem Class

An important precondition of gaining generalizable knowledge is a well defined problem class on which algorithms can be compared [4]. For the purpose of this paper we use the subset sum problem as an example. It is an NP-hard combinatorial optimization problem. Besides, it is known where the hard instances lie [1].

In the subset sum problem we are given a set $W = \{w_1, w_2, \ldots, w_n\}$ of n positive integers and a positive integer M. The task is to find a $V \subseteq W$ such that the sum of the elements in V is closest to, without exceeding, M. For this problem it is possible to define a problem instance generator. Let us assume that a triple (n, k, d) is given where n is the set size as above and $d \in [0, 1]$ is called the *density* of the problem. The set W is created by drawing w_i randomly from the interval $[0, 2^k]$ with a uniform distribution for $i = 1, \ldots, n$ and let M be the sum of a random subset of W of size $[nd]$. We will denote the generated problem class by SubSum(n, k, d). Our running example in the paper will be SubSum$(100, 100, 0.1)$.

It is essential that the problems in a problem class exhibit some common structure in some sense, some similarity that can be exploited and converted to durable knowledge. Note that structure as meant here arises from a combination of an algorithm and a problem class. For a fixed algorithm space the problem class is structured (non-random) if the behavior (performance) of the different algorithms shows some regularities on different instances from the problem class. The lack of such regularities would indicate

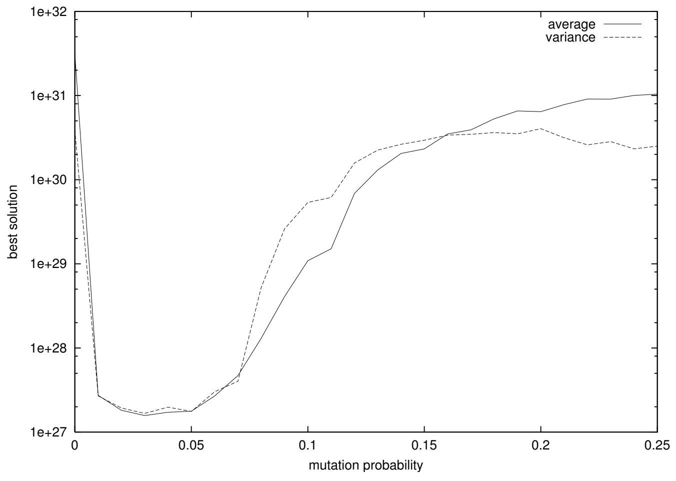

Fig. 1.

that the algorithm space in question and the problem class are not related: there is nothing to be learned.

The graph in Figure 1 shows such regularities here with respect to a simple (1+1) EA. The EA uses a binary representation with every bit representing the presence of a given set element in the candidate subset, and a a bit-flip mutation. Different EAs correspond to different mutations, defined by the mutation rate. Figure 1 plots the performance of such mutation rates, defined as the distance from the desired sum (to be minimized) of their "hosting" EA. Each point in the curve was generated using 100 runs on different instances, all runs until 10000 evaluations. The instances for different mutation probability values were different as well.

Figure 1 provides a confirmation that there is a link between mutation operators and algorithm quality: there are some regularities to be exploited, there is something to learn. It is important to note that this is not necessarily all the regularity we can find. Therefore, we are also interested in other algorithms that might exploit other regularities. The very essence of our approach is that we try to explore and discover as much similarity as possible, using other, richer algorithm spaces.

2.2 An Example Algorithm Space

In the case of SubSum$(100, 100, 0.1)$ let us fix the (1+1) EA applied until 10000 evaluations with the binary representation mentioned above. The algorithm space we will use is given by the mutation operator which is defined by two parameters: p_{01} defines the probability of flipping a 0 to 1, and p_{10} defines the probability of flipping a 1 to 0. If $p_{01} = p_{10}$ then we get the traditional bit flip mutation. This choice of representation involves domain knowledge as well [9]. Since the density of the problem is relatively low, it might be better if the operator can express a bias towards solutions that contain more 0s than 1s. Our mutation operator can express this bias.

3 A Suitable Tool: DRM

3.1 DRM Structure

The interested reader is kindly asked to refer to [11, 12] for a detailed description of the DRM. A main feature of the DRM is that applications are implemented on it in the conceptual framework of multi-agent systems. In fact, the agents are the threads of the distributed applications. They are mobile, they can communicate with each other and they can make decisions based on the state of the DRM or the application they participate in.

As applications are multi-agent applications, the main task of the DRM is to support these agents. The framework is implemented in Java language which provides mobility and security. Furthermore, to maximize scalability and robustness, the DRM is a pure P2P system, which relies mainly on epidemic protocols [2]. This means that the computers participating in a DRM know only a limited number of other computers from the same DRM. Via exchanging information with these peers only, information spreads as gossip (or epidemic) through the DRM.

3.2 Experiment Structure

To run an application on the DRM one has to think in terms of a multi-agent system. Designing an experiment involves designing the behavior of one or more types of autonomous agents that should perform our experiment. The main concerns in our case are robustness also at the experiment level, not only at the level of the DRM, and the mechanism of distributing information and computation over the DRM.

Our parallelization approach is roughly based on island models. That is, every agent hosts an island where a local population is evolved. The major challenge is however to design a communication mechanism that is robust and scales well in a wide area network environment.

Island Communication The overall structure of the proposed design mirrors the structure of the DRM itself. We use completely identical islands that communicate using epidemic protocols. Note that it is not an evident design decision; even though the DRM itself is pure P2P, it is possible to design an agent system where one of the agents plays the role of the server of the experiment while the others are the clients.

Our decision is based on our and others' recent findings that indicate the power of our epidemic protocol concerning scalability and robustness [12]. These features are most important in a highly distributed environment like the DRM where network communication goes over a wide-area network and the number of available machines is not known in advance.

At first, the root island is started up by the experimenter who also provides a maximum number of tasks to be computed as a parameter. Each task is an island which is supposed to run the high-level learning EA until a given termination criterion.

Note that the actual number of involved machines depends on their availability and reachability from the root island. The latter is influenced by network separating devices like firewalls between the available machines. Due to the utilized P2P technology, machines do not necessarily need a direct path to each other. Information spreads like an epidemic throughout the DRM in an undirected manner via a chain of islands to finally reach the destination.

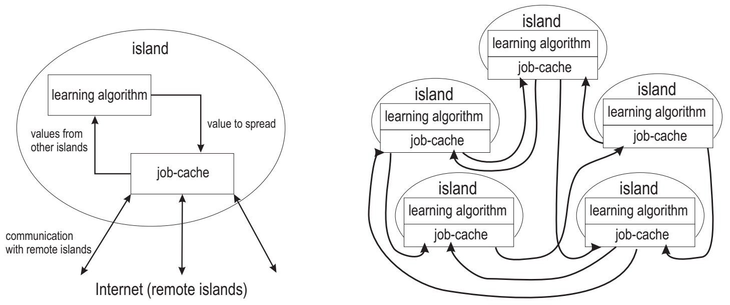
island
learning algorithm
value to spread
values from other islands
job-cache
communication with remote islands
Internet (remote islands)
island
learning algorithm
job-cache

Fig. 2. left: learning algorithm and job-cache communicating within an island, right: example of an incompletely connected set of islands running the same experiment

Task distribution is done by self-replication of islands according to the load balancing algorithm laid out in [11]. In short, an island that encounters an empty machine sends half of its own tasks there. In contrast to other possible scenarios of parallel experiment startup, we distribute tasks dynamically during experiment run.

Just like the DRM layer which utilizes incomplete databases for node communication, the islands have an incomplete experiment database called the *job-cache*. Every job-cache entry holds the address of another island and its attached value. The address/value table is merged with the one of a remote island by exchanging messages using a randomly selected address and cut to the defined maximum database size. While the expected number of islands doing an experiment may be lower than the number of DRM nodes and thus the job-cache can be smaller, it inherits the advantageous properties of the DRM database, namely information spreading speed and the very low probability of the experiment getting partitioned. Algorithms running within an island may put data into this repository and regular job-cache message interchange will spread these values to other islands in logarithmic time. Figure 2 illustrates messaging between the algorithm and the job-cache (left) and the interconnection of islands via their job-caches (right). For experiments with many islands, these do not necessarily know all other islands because values are not only transferred directly but also indirectly by traversing several job-caches.

Prerequisites of Learning Learning a good mutation operator for our example, SubSum(100,100,0.1), while varying only two parameters (p_{01} and p_{10}) may not seem very hard, but a first analysis utilizing a grid search over the most interesting area (see Figure 3) reveals at least three difficulties. First, evaluations of one specific operator on different problem instances display a dangerously high variance. The learning algorithm therefore has to average the results of several runs. Second, the approximated fitness landscape as depicted in Figure 3 clearly indicates a wide, almost flat area for $p_{01} > 0.25$. A path-based optimization method depends on a good start point if the neighborhood of the actual solution does not offer any progress. An exploration-based technique can solve this problem although it typically needs more evaluations. Third, the parameter p_{10} has only minor influence on the fitness of the operator for most regions. It increases the search-space but gives little help to finding good solutions if the first parameters value is too high.

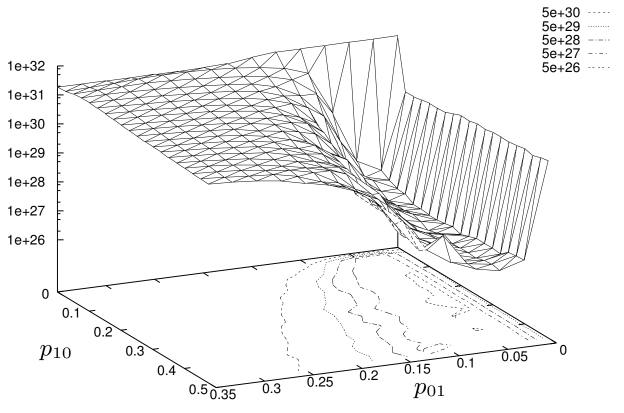

Fig. 3. Performance of the operator described in Section 2.2. Average of 100 runs (each on a new randomly generated problem instance from $\mathrm{SubSum}(100, 100, 0.1)$) per point.

To summarize, a successful learning algorithm for this problem class has to be able to cope with a very noisy function and should apply exploration as well as local search.

Learning Algorithm Based on these prerequisites, we chose a population-based EA as learning algorithm. In a parallel environment there are at least two ways of implementing a population. First, every island may run a pure local search algorithm and integrate search results of the remote islands from time to time. Second, each island may keep a population on its own and add remote results as they become available. Concerning the DRM, the second approach is favorable because wide-area network communication is unreliable and rather slow. Nevertheless, both methods have been tested. To prevent the search from getting stuck, the employed selection/replacement operator does not allow for old solutions to survive and thus equals EA comma-selection.

As described above, islands can distribute preliminary results to their neighbors by putting them into the job-cache. Solutions exchanged in our case are the best mutation operators available in the local population.

In the following, we discuss the details of the learning algorithm from the viewpoint of a single island. Every new generation starts with reading the available remotely computed solutions from the job-cache and merging them with the current parent population. Offspring is generated by each time choosing two solutions from the result and applying intermediate recombination, followed by Gaussian mutation. We evaluate the newly created mutation operator by selecting the appropriate (1+1) EA from the defined algorithm space and applying it 5 times to randomly generated instances of the SubSum(100,100,0.1) problem class.

The next generation's parent population is then constructed by selecting the best mutation operators found in the offspring. Note that each solution sent by another island will influence the learning process until replaced by its originator. This rule resembles a *shared memory* concept known from parallel hardware and differs significantly from migration. As a consequence, new islands starting up inherit a set of solutions with acceptable quality after connecting to any other island.

	single machine $(1,5)$	single machine $(4,20)$	DRM $(1+m,5+4m)$	DRM $(4+m,20+4m)$
best fitness	1.67e26	1.56e26	1.92e26	2.038e26
best evaluation no.	640	682.6	451.5	521.5
concurrent islands	1	1	3.9	4.2
AES	240	232.5	187.5	195.5
success rate	50%	80%	100%	100%
best operator p_{01}	0.066	0.048	0.048	0.066
best operator p_{10}	0.634	0.515	0.545	0.589

Table 1. Accumulated results for four different types of experiments

Let m denote the number of remotely computed solutions available. As long as $m = 0$, the learning algorithm equals a (μ, λ) ES, μ and λ meaning the sizes of the local parent population and the offspring, respectively. In the experiments we used $\lambda = 5$ and $\mu = 4$. However, as soon as solutions from remote islands arrive, (μ, λ) changes to a $(\mu + m, \lambda)$ ES. To keep up selection pressure and thus ensure a good local search capability we also increase the number of offspring solutions created within a generation by a factor λ_r (=4), resulting in a $(\mu + m, \lambda + \lambda_r m)$ ES. Note that the upper bound of m equals the maximum number of entries of the job-cache.

A possible danger is that bad solutions may remain in the job-cache if islands become unavailable due to e.g. network failure. However, for a realistic scenario this is not true on the long run because usually the number of tasks exceeds the size of the job-cache so that old information is gradually removed from the system.

3.3 Experimental Results

Table 1 shows the accumulated results for four types of experiments, namely the $(1, \lambda)$ and (μ, λ) ES on a single machine and the $(1 + m, \lambda + \lambda_r m)$ and $(\mu + m, \lambda + \lambda_r m)$ ES running on the DRM. All values are averages over 10 runs per experiment type. The *best fitness* measures the quality of the best mutation operator in terms of the best subset sum solution found by the (1+1) EA using this operator. Let us note that according to the t-test the differences between the values of the best solutions for the four algorithms are not significant (for $\alpha = 0.05$). *best evaluation no.* gives the number of root island evaluations needed to generate the best solution. As mentioned in the previous section 5 tests are done with each operator variant so the number of mutation operators tested is $best\ evaluation\ no./5$. From the grid search depicted in Figure 3 we knew in advance that mutation operators with an average fitness below $1e27$ exist and are already near the optimum. We therefore computed the *success rate* as the fraction of experiments with a best fitness below this upper bound. The *average number of evaluations to solution (AES)* gives the number of root island evaluations needed to reach this level. The last two rows show the found mutation operator parameters p_{01} and p_{10}.

The best results are found sooner on the DRM, this is a side-effect of the fact that many populations work in parallel. The table does not directly reveal the actual speedup of the system. However it was proven in [11] that the speedup is almost linear for any application that is distributed the way our present application was.

The optimal mutation parameters found by the algorithm show a consistent pattern. The ratio p_{01}/p_{10} is close to 0.1 which is as a matter of fact the density parameter of the problem class SubSum(100,100,0.1). It is also notable that the mutation operator corresponding to the optimal parameters outperforms the optimal one-parameter mutation. This can be seen clearly by comparing the value of the best solutions found to the performance of the one parameter mutation shown in Figure 1. This illustrates the possible benefits of learning more complex operators.

Even in the case of our simple example when the operator space was defined by two real parameters it is quite clear that searching this space is much more effective with the automatic approach than using an exhaustive search. To illustrate this, consider that visualizing the whole $[0, 1]^2$ space in the quality shown in Figure 3 would require 160000 runs of the underlying (1+1) EA. Even this quality is hardly enough to draw any consistent conclusions over the location of the global optimum due to the high variance. Compared to this our DRM application needed only around 500 runs to learn a very good quality operator. It is of course not surprising that evolutionary search is much better than exhaustive search. This remark is useful only to point out that this quite trivial fact holds in the case of operator learning as well, which is another argument for the automatic acquisition of durable knowledge.

Finally, let us note that all experiments have been run from behind a firewall, using a 56k dial-in network connection. This may increase island distribution time and block several messages directly targeted at the root island but demonstrates the applicability of the DRM technology even under the worst conditions.

4 Conclusions and Future Perspectives

In this paper we have argued for the importance of learning algorithm components for problem classes. The durable knowledge that results from this process can be directly applied to improve algorithms or it can be analyzed further to gain scientific insight into a problem class. A tool was suggested that is suitable for performing this kind of expensive learning by utilizing the idle time of any computers connected to the Internet. The feasibility of the approach was demonstrated on the problem class SubSum(100,100,0.1).

An interesting possibility for future applications is the possibility of developing self-improving software packages. The different copies of the software could communicate through the Internet and exchange information with the help of the DRM or a similar P2P environment that uses epidemic protocols for communication. This process can be completely transparent to the user and due to the P2P approach it could scale very well to millions of copies without any special investment. At this level a learning algorithm could gradually improve the performance of the software via evolving durable knowledge based on the experience on the problems people are trying to solve with the software. Note that this application area is not restricted to learning heuristic operators. An arbitrary component of a software can be improved provided an appropriate evaluation method is available.

Acknowledgments

The authors would like to thank the other members of the DREAM project for fruitful discussions, the early pioneers [13] as well as the rest of the DREAM staff, Maribel García Arenas, Emin Aydin, Pierre Collet and Daniele Denaro.

References

1. M. J. Coster, A. Joux, B. A. LaMacchia, A. M. Odlyzko, C.-P. Schnorr, and J. Stern. An improved low-density subset sum algorithm. *Computational Complexity*, 2:111–128, 1992.
2. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database management. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, Vancouver, Aug. 1987. ACM.
3. A. E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, July 1999.
4. Á. E. Eiben and M. Jelasity. A critical note on experimental research methodology in EC. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC 2002)* [8], pages 582–587.
5. A. E. Eiben and C. A. Schippers. Multi-parent's niche: n-ary crossovers on NK-landscapes. In W. Ebeling, I. Rechenberg, H.-P. Schwefel, and H.-M. Voigt, editors, *Parallel Problem Solving from Nature - PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 319–328. Springer-Verlag, 1996.
6. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.
7. J. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128, 1986.
8. IEEE. *Proceedings of the 2002 Congress on Evolutionary Computation (CEC 2002)*. IEEE Press, 2002.
9. M. Jelasity. A wave analysis of the subset sum problem. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 89–96, San Francisco, California, 1997. Morgan Kaufmann.
10. M. Jelasity. Towards automatic domain knowledge extraction for evolutionary heuristics. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VI*, volume 1917 of *Lecture Notes in Computer Science*, pages 755–764. Springer-Verlag, 2000.
11. M. Jelasity, M. Preuß, and B. Paechter. A scalable and robust framework for distributed applications. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC 2002)* [8], pages 1540–1545.
12. M. Jelasity, M. Preuß, M. van Steen, and B. Paechter. Maintaining connectivity in a scalable and robust distributed environment. In H. E. Bal, K.-P. Löhr, and A. Reinefeld, editors, *Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002)*, pages 389–394, Berlin, Germany, 2002. IEEE, IEEE Computer Society.
13. B. Paechter, T. Bäck, M. Schoenauer, M. Sebag, A. E. Eiben, J. J. Merelo, and T. C. Fogarty. A distributed resoucre evolutionary algorithm machine (DREAM). In *Proceedings of the 2000 Congress on Evolutionary Computation (CEC 2000)*, pages 951–958. IEEE, IEEE Press, 2000.
14. M. Pelikan, D. E. Goldberg, and F. Lobo. A survey of optimization by building and using probablistic models. Technical Report 99018, Illinois Genetic Algorithms Laboratory, 1999.
15. R. G. Reynolds. Cultural algorithms: Theory and applications. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, Advanced Topics in Computer Science, pages 367–377. McGrow-Hill, 1999.
16. SETI@home. http://setiathome.ssl.berkeley.edu/.
17. A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.
18. United Devices$^{\text{tm}}$. http://ud.com/.