Statistical Analysis of Heuristics for Evolving Sorting Networks

Lee Graham Hassan Masum Franz Oppacher

Carleton University
1125 Colonel By Drive
Ottawa, Ontario, K1S 5B6

Contact: {lee@stellaralchemy.com, hmasum.com, oppacher@scs.carleton.ca}

ABSTRACT

Designing efficient sorting networks has been a challenging combinatorial optimization problem since the early 1960's. The application of evolutionary computing to this problem has yielded human-competitive results in recent years. We build on previous work by presenting a genetic algorithm whose parameters and heuristics are tuned on a small instance of the problem, and then scaled up to larger instances. Also presented are positive and negative results regarding the efficacy of several domain-specific heuristics.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning – *parameter learning*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search - *heuristic methods, graph, and tree search strategies*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation – *unbounded-action devices*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems – *sorting and searching, computations on discrete structures*; G.2.1 [**Discrete Mathematics**]: Combinatorics – *combinatorial algorithms, permutations and combinations*;

General Terms

Algorithms, Performance, Experimentation, Design, Theory, Measurement

Keywords

Sorting Networks, Genetic Algorithms, Parameter Tuning.

1. INTRODUCTION

Sorting is a fundamental operation in computer science, with algorithms ranging from straightforward to highly complex. The challenge is to devise an algorithm that is guaranteed to arrange an input set of data items according to some given ordering function, while minimizing quantities like total time or (for large data sets) space used.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'05, June 25-29, 2005, Washington, DC, USA.
Copyright 2005 ACM 1-59593-010-8/05/0006…$5.00.

Sorting networks are a particular instantiation of sorting algorithms that are particularly suited for implementation in hardware. The basic element of a sorting network is the "comparator". Each comparator has two input lines and two output lines. Two items enter the input lines in arbitrary order; the comparator then orders the pair of items, and outputs the lesser one on the first output line and the greater one on the second output line.

By chaining together sets of comparators into a sorting network, any set of items can be sorted. The Zero-One Principle guarantees that if a sorting network can sort all possible binary input vectors (or "bit-lists") then it can also sort any arbitrary input vector. Thus, a "brute-force" method of checking the validity of a sorting network with **N** input lines is to try it on all $\mathbf{2^N}$ bit-lists of size **N**, and ensure that the output is correctly sorted in all cases. (Note that imperfect sorting networks can be ranked by the number of bit-lists which they correctly sort – herein referred to as the network "score")

Knuth's classic volume [7] discusses sorting networks in some detail, and in particular records lower and upper bounds for the best known sorting networks. There are two natural metrics for judging sorting networks. First, the number of comparators used. Second, the "depth" of the network – this is the number of time steps the network takes to complete its computation, if an arbitrary number of comparators operating on disjoint elements are allowed to compute in parallel at each time step. Each such group of disjoint comparators executing in parallel may be referred to as a "level" in the network.

Since the problem of designing sorting networks is theoretically challenging, of practical interest, and of some long-lasting notoriety in algorithm design circles, it makes a good test problem for evolutionary computation. [5] is an early paper which used co-evolutionary methods to evolve both sorting networks and bit-lists which were particularly good discriminators for the quality of those networks. Co-evolutionary methods were more recently used to evolve fault-tolerant sorting networks [4]; theoretical approaches have also yielded interesting strategies for making a sorting network robust, such as adding a fault-fixing layer after the main network [9]. There have been several other papers using a variety of evolutionary strategies [1, 2, 3, 6, 8].

In this paper, we make several new contributions to this area. First, we discuss the methodology, network representation, and the parameters of the GA. We use statistical methods to get confidence intervals on the value of parameter settings and heuristic choices. Second, we design several heuristics specialized for the problem of evolving sorting networks, and evaluate these heuristics to learn which ones improve solution quality. Third, we show that the GA

performance improves significantly with tuning, and we apply the tuned parameters to larger instances of the problem.

All of our contributions are validated through testing on networks of various sizes, from 10 to 14 elements. Some of our results match other records for best known sorting network results (some of which were found through human ingenuity, others through computational search). A noteworthy feature is that our heuristics are effective enough that all of our computations could be carried out on ordinary PCs, unlike several previous evolutionary computation results which used large clusters of workstations.

2. THE GENETIC ALGORITHM

This section explains the approach taken by the authors in terms of a breakdown of the important aspects, parameters, and genetic operators within the GA. Also explained is the manner in which the various parameter values were settled upon during a tuning phase for the GA, and then scaled up to larger instances of the problem.

2.1 Methodology

The approach taken in this experiment is to run a tuning phase for the GA on a small problem instance – in this case 10-input sorting networks – and thereafter scale up to 12-, and 14-input networks.

The tuning phase consists of two steps. The first step is to optimize the GA parameters such as crossover rate, mutation, etc. Even given a coarse-grained discretization of those parameters whose ranges can be thought of as continuous – specifically probabilities ranging from 0.0 to 1.0 – the number of possible settings is too large for an exhaustive search of parameter space. The authors decided to tackle the optimization problem one parameter at a time beginning from an arbitrary starting point. The criterion used when optimizing a given parameter is to choose that value for which the average best individual after 200 generations, over multiple runs, is highest. In all cases a minimum of 50 runs per parameter value are performed, more if the corresponding confidence intervals are too wide for a reasonably near-optimal setting to be distinguished. Once a parameter is optimized in this way its value becomes fixed and the process moves on to the next as yet untuned parameter.

The second step in the tuning phase is to introduce each of four additional domain-specific heuristics which modify the sorting network representation and/or the fitness function. Along with each heuristic come one or more additional parameters which are tuned in a manner similar to that described in step 1.

Since perfect networks (networks sorting all 1024 bit-lists) become more common once the GA is tuned, we abandon average best-of-run as a measure of goodness and focused instead on success rate. The following calculation is performed as a measure of goodness, **G**:

$$\mathbf{G} = (\mathbf{N}_{1015} + \mathbf{N}_{1016} + \ldots + \mathbf{N}_{1022} + \mathbf{N}_{1023}) / 9$$

Where **Ni** is the number of runs attaining a network score greater than **i**. This is a linear weighting whereby each run attaining a score of 1016 contributes equally to **G**, runs attaining a score of 1017 contribute twice as much, 1018's thrice as much, and so on. The rationale behind this equation stems from the intuition that the rate of attainment of near-perfect scores should act as a useful predictor of achievement of perfect scores. In order to validate this intuition empirically, the authors collected data from 149 different parameters settings of the GA during tuning, each represented by at least 50 independent runs, and for scores, **i**, from 1002 to 1023, calculated **r** – the coefficient of correlation with success rate.

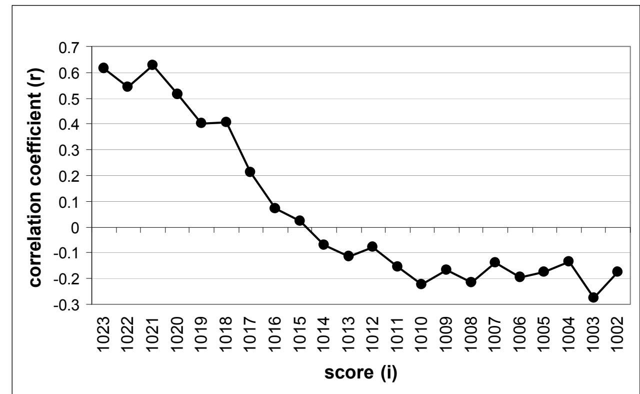

Figure 1. Correlation coefficient of score rates with success rate based on 149 different GA parameter settings

A graph of **i** vs. **r** (see figure 1) shows the relationship to be close to linear over the range used to compute **G**, above. This result lends support to the use of **G** as a measure of goodness for GA parameter settings when raw success rate itself is low.

Once GA parameters and heuristics have been optimized, the problem size is scaled up from 10-input networks to 12-, and 14-input networks, with corresponding increases in population size and number of generations.

2.2 Representation

Networks are stored as fixed-length arrays of comparators. Each comparator is specified by two integers indicating which lines of the network partake in the corresponding comparison-exchange operation. For example, the 7-input network in figure 2 may be represented as [[0,1], [2,3], [4,5], [6,7], [1,3], [5,7], [2,4], [0,6], [1,2], [3,4], [5,6]]. Inputs (a sequence of integer values, one for each horizontal line) arrive at the left and travel through the network to the right, undergoing sorting operations along the way. Each sorting operation examines two inputs and outputs them in sorted order (a potential swap).

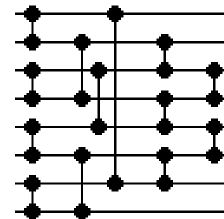

Figure 2. A sorting network

Most of the sorting networks given in Knuth [7] display a symmetry in their structure in which many comparators, especially those in the initial levels of the networks, are paired with their mirror image (the same comparator, but with network input lines renumbered in reverse order). Assuming that this reflects a tendency for such symmetric networks to be superior to asymmetric networks – an assumption tested empirically and shown to be correct via a clear distribution-lensing effect in figure 3, based on over 4,000,000 samples – the population is initialized with networks for which

every comparator is paired with its mirror image (provided the comparator is not its own mirror image in which case a second copy is redundant).

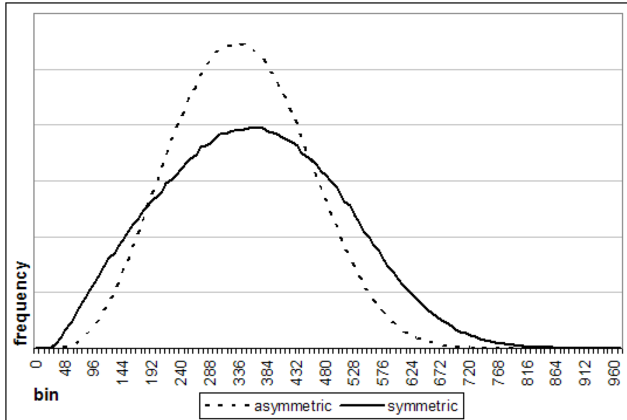

Figure 3: Score distributions for random asymmetric and symmetric networks

This symmetry is enforced only in the initial population, but not in subsequent generations. Note also that due to a conflict between this symmetric network initialization for networks with odd numbers of inputs, and a particular heuristic, to be described in section 3.4, the authors have restricted the experiments described here to networks with even numbers of inputs.

2.3 Parameters

The parameters described below are those that pertain to the functioning of the GA. Many of these, such as population size, mutation rate, and tournament size, are common to most genetic algorithms, while others are specific to this implementation.

2.3.1 Population Size and Halting Condition

For runs on 10-input sorting networks, population size and number of generations are fixed at 500 and 200, respectively. Once a network with perfect score is attained, there still remains the possibility for improvement – by finding a network of lesser depth – and hence a fixed number of generations is left as the only halting condition.

2.3.2 Fitness, Elitism and Selection

Network fitness is computed by counting the number of bit-lists (out of $\mathbf{2^N}$ possible, where **N** is the number of inputs) which are sorted after a pass through the network. In the second phase of tuning, in which a handful of heuristics are introduced, the fitness calculation is modified by subtracting various penalties from this initially-computed network score.

The GA implements elitism by maintaining, from one generation to the next, not the highest fitness individual but the highest scoring individual. For most runs network fitness and network score are synonymous. It is only in those runs using the heuristics described in section 3 that fitness can differ from network score. While fitness guides selection, we decided that elitism should maintain the highest scoring network regardless of its fitness value. Ties are broken using network depth, with lower depth networks being superior.

The selection method chosen is tournament selection with both tournament size and tournament probability as parameters to be optimized. The best individual in a tournament is determined by fitness, as described above.

Network duplicates are filtered out when producing a new generation. This is done via a hash function since this is much faster than computing network score.

2.3.3 Crossover

Two crossover operators are made available to the GA, uniform crossover at the level of individual comparators, and single-point crossover. This accounts for two GA parameters. The first is the usual crossover probability parameter; the second determines the proportion of uniform vs. single-point crossover.

2.3.4 Mutation

Three mutation operators are provided. The first, called 'indirect replacement', chooses a comparator to delete from the network, and then chooses an insertion point at which to place a new random comparator keeping the total count fixed. The second chooses two comparators at random and swaps them. The third performs point mutation on up to three consecutive comparators. These operators necessitate the usual mutation rate parameter, plus additional parameters determining the probabilities for each mutation operator. Interestingly, the best results were obtained when mutation was restricted exclusively to indirect replacement.

3. HEURISTICS

The following subsections describe 4 intuitively useful heuristics for guiding the GA evolution and constraining its search space. The results obtained during testing indicate that, despite their intuitive plausibility, only one of the four heuristics appears to be useful in practice.

3.1 Target Depth and Depth Penalty

This heuristic established a target network depth, specifically the depth of comparison-optimal (having minimal comparator count) networks as given in Knuth [7]. Networks are penalized for exceeding this depth by an amount which is proportional to the total number of comparators in the **k** smallest levels where **k** is the difference between network depth and target network depth. The purpose of this heuristic is to steer GA evolution toward a desired network depth, and to do so in a manner that rewards intermediate steps between a network of depth **d**, and a network of depth **d-1**.

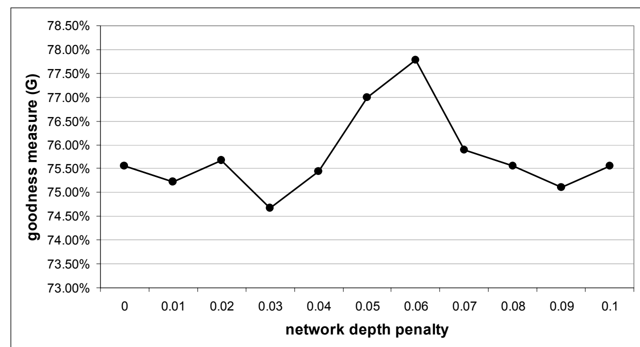

Figure 4. Network depth penalty vs. goodness (G) for depth penalty heuristic

In figure 4, a graph of network depth penalty vs. goodness (see section 2.1) leads the authors to choose a value of 0.06 for this parameter, which translates to a penalty of $0.06 * 2^{10} = 61.44$ subtracted from fitness for each comparator in the **k** smallest levels as described above. It should be noted that despite the appearance of figure 4, the range of outcomes for this parameter in terms of average best-of-run values span a range of less than one bit-list out of 1024. 650 runs per parameter value were used in calculating each goodness value in figure 4.

3.2 Redundant Comparator Penalty

This heuristic penalizes networks for the possession of redundant comparators. A redundant comparator is one whose removal does not affect the performance of the network. This is easily determined by counting the number of times a comparator performs an exchange operation while the network is attempting to sort all 1024 bit-lists. Comparators for which this total is 0 are redundant. Networks containing redundant comparators are penalized in proportion to the number of such comparators present.

While the argument for penalizing redundant comparators is obvious, it is conceivable that rewarding redundant comparators might also be beneficial, in that one can imagine that such a scheme might force the remaining non-redundant comparators to be as useful as possible. For this reason we experimented with negative values of this parameter as well as positive, calculating the goodness measure using 400 runs for each of 15 values of the penalty parameter between -0.1 and 0.1. Counterintuitively, however, the testing results obtained seem to indicate that this heuristic is harmful, both as a penalty and as a reward.

3.3 Extreme Value Penalty

This heuristic penalizes networks for all instances in which extreme values at the inputs do not end up at the correct outputs. In the case of 10-input networks, this is determined by examining the outcome from attempts to sort 20 specific bit-lists, those containing nine 0's and a 1, and those containing nine 1's and a 0. The rationale behind this heuristic is that in the general case where a network is sorting inputs of arbitrary numerical values instead of being restricted to bits, these extreme value bit-lists correspond to a tenth of all inputs in the general case, but only a 1024th of all inputs in when using bit-lists. This heuristic is meant to place added value upon these bit-lists, but like the redundant comparator penalty it too was found to be harmful. 300 runs went into the calculation of each goodness measure for this heuristic's parameter.

3.4 Full Level Enforcement

Many of the sorting networks given in [7] possess initial levels which contain the maximum number of comparators, specifically **N / 2** disjoint comparators where **N** is the number of network inputs. This seems to suggest an obvious heuristic, which is to modify network representation and manipulation such that initial levels are guaranteed at all times to be 'full' (possessing **N / 2** disjoint comparators). This is achieved via appropriate modifications to crossover and mutation operators.

Mutation within one of these full levels (point-mutation in this implementation) is followed by a repair operation. Similarly, crossover operators which disrupt a full level are followed by the same repair operation. This heuristic adds to the number of crossover and mutation operators as well. Crossover is now split into two operations, the first a crossover of initial full network levels – either uniform crossover of entire levels, or single-point crossover within levels, followed by a repair operation – and the second a crossover of the remaining comparators as described in section 2.3.3. Mutation is similarly split into two operations, mutation of full network levels (either point mutation or comparator reordering) and mutation of the rest of the network as described in section 2.3.4.

New parameters corresponding to these new mutation and crossover operators were also subject to optimization, but like the redundant comparator penalty and the extreme value penalty, this heuristic too was found to be harmful. 400 runs per goodness calculation were used for this heuristic.

4. SCALING UP

The GA tuning began with arbitrary settings for its parameters, and with none of the heuristics from section 3 (see table 1). Its initial performance, as measured by average best-of-run network score, was between 1013.50 and 1013.91 (a 95% confidence interval) with a standard deviation of 4.04. Its outcome distribution is shown in figure 5.

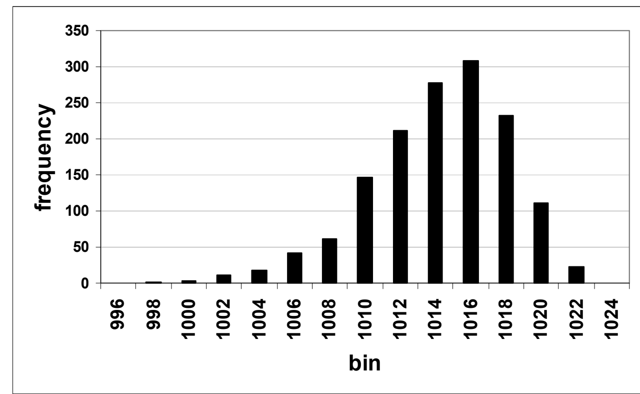

Figure 5. Best-of-run bin distribution for initial GA settings

Upon completion of tuning, GA performance was between 1018.58 and 1018.90 (95% confidence interval) with a standard deviation of 3.02. Its outcome distribution is shown in figure 6, and the corresponding optimized parameters are shown in table 1. Figures 5 and 6 display the same set of bins for easy comparison.

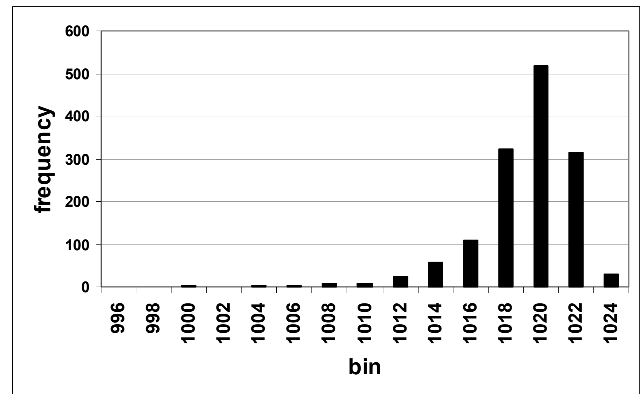

Figure 6. Best-of-run bin distribution after tuning

Table 1. GA parameters before and after tuning

	before	after
Population size, Generations	500, 200	500, 200
Crossover rate	20%	10%
Crossover operators	50% uniform, 50% single-point	30%, 70%
Per Network Mutation rate	10%	40%
Mutation operators	33%, 33%, 33%	100%, 0%, 0%
Tournament size	4	8
Tournament probability	80%	100%
Elitism priorities	Score. Depth breaks ties	Score. Depth breaks ties
Depth penalty	0.0	0.06 (61.44 out of 1024)

The distribution of outcomes clearly shows the superior performance of the GA after tuning, indicating that the tuning process was responsible for a substantial performance boost. We can also see this by looking at the rate at which networks which perfectly sorted all bit-lists were found. The success rate for the GA before tuning was 0 out of 1450 runs, and after tuning it was 31 out of 1400 runs.

4.1 10-Input Networks

The results for 10-input networks are given above. During 'tuning' many dozens of optimal networks were produced, such as the one shown in figure 7, below. This is a 29-comparator, depth-8 network. It is actually better than the best network given in Knuth [7], and hence would have been a record-breaker back in 1998 (others have found equivalent networks in the last several years).

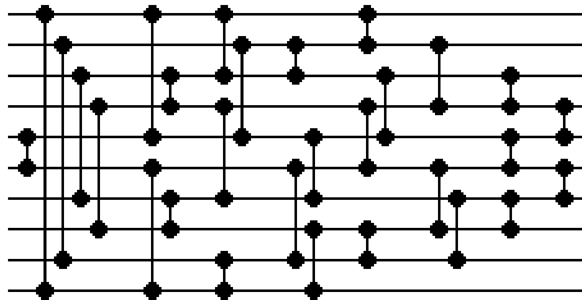

Figure 7. One of many optimal 10-input sorting networks discovered by the GA

4.2 12-Input Networks

When scaling up to a more difficult instance of the problem there arises the question of exactly how to perform the scaling. In this experiment all but two GA parameters are held constant in the step up from 10-input to 12-input networks. The two parameters modified were population size and number of generations, each of which was doubled, quadrupling the number of network evaluations per run to reflect the fourfold increase in problem difficulty as measured by the number of bit-lists to sort. Whether or not this is an appropriate approach is certainly open to further experimentation. Note also that the network depth penalty (see section 3.1) is scaled up automatically. The chosen parameter value of 0.06, which had previously translated to a penalty of 61.44, now becomes $0.06 * 2^{12} = 245.76$.

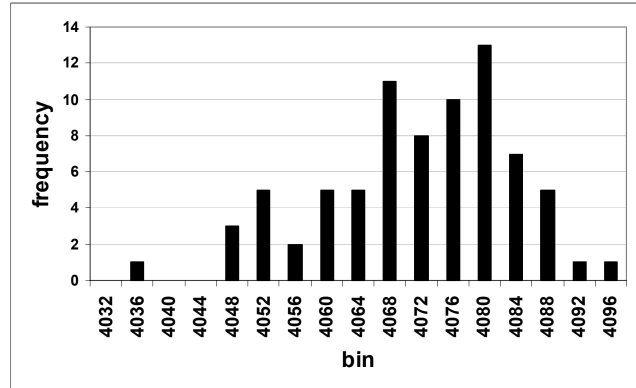

Figure 8. Best-of-run bin distribution for 12-input networks

The outcome distribution for the 12-input problem is shown in figure 8, above. The average best-of-run outcome is between 4065.73 and 4072.52 (95% confidence) with a standard deviation of 15.3. A single success was achieved out of 78 independent runs, and the resulting optimal network is shown in figure 9. This is a 39-comparator, depth-9 network, equivalent to the best network given in Knuth [7].

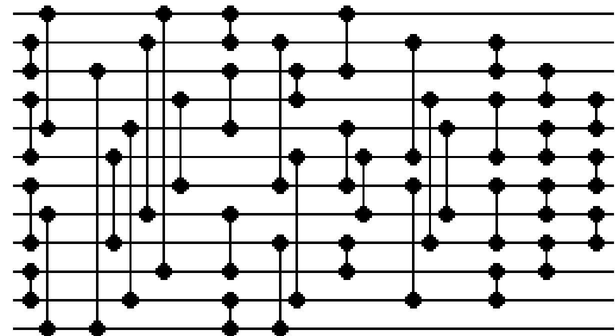

Figure 9. An optimal 12-input sorting network discovered by the GA

4.3 14-Input Networks

81 runs were collected for the 14-input problem, and population size and number of generations were doubled yet again to reflect the increased problem difficulty. In this case no success was achieved, although the outcome distribution shown in figure 10 seems to hint at the possibility that more runs may yield a success. The average best-of-run network score was between 16311.84 and 16330.21 (95% confidence interval) with a standard deviation of 43.72.

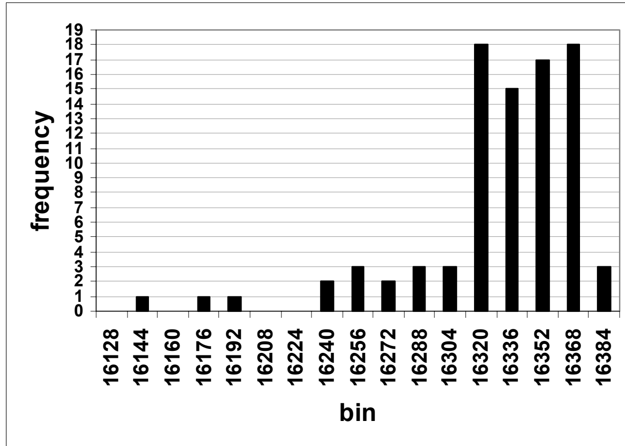

Figure 10. Best-of-run bin distribution for 14-input networks

5. CONCLUSION

In this paper, we have done careful parameter tuning and statistical analysis of heuristics for evolving sorting networks. This has provided both positive and negative results with respect to the value of several natural heuristics.

We found that, of the four heuristics tested, only depth penalty showed benefits. The three other heuristics, despite their intuitive plausibility, did not stand up to rigorous testing. This may be seen as an object lesson in the value of carefully testing plausible-seeming heuristics individually, instead of only testing the success of all heuristics in combination.

We have also implemented an efficient system, which has matched several previous sorting network records while using only modest computational resources.

Finally, the statistical analysis on best-of-run distributions is a useful tool for understanding how well heuristics work, and for predicting the average waiting time to achieve optimal results. Calculating correlation coefficients of near-perfect run outcome values with optimal outcome values validated the use of near-perfect run outcomes as part of a candidate measurement of GA performance in determining optimal parameter values, especially in cases for which perfect outcomes are rare.

There are several natural directions for future research. We believe that other heuristics may provide better outcomes, which will be a particularly important consideration for larger network sizes. To scale up to networks beyond about 16 inputs, new approaches will be needed - starting perhaps with “progressive sampling” schemes where a co-evolved testing set of bit-lists similar to that used in [5] is used in a first pass, and successively more bit-lists in later passes. Structural approaches [2, 3] as well are clearly important for reducing the search space.

Sorting networks have been a fascinating computer science challenge for several decades. Since the early 1990's, evolutionary computing methods have managed to match several of the best-known results for moderate network sizes. It will be interesting to see if evolutionary computing methods can be scaled up to break the records for large network sizes. We would also suggest that this problem provides a good test-bed for comparing algorithm performance, and for carefully studying evolutionary heuristics.

6. ACKNOWLEDGMENTS

We would like to thank Steffen Christensen for several helpful suggestions.

7. REFERENCES

[1] Sung-Soon Choi and Byung-Ro Moon. *A New Approach to the Sorting Network Problem Evolving Parallel Layers*. In Proceedings of GECCO-2001. Morgan Kaufmann, 2001, pp. 258-265.

[2] Sung-Soon Choi and Byung-Ro Moon. *Isomorphism, Normalization and a Genetic Algorithm for Sorting Networks*. In Proceedings of GECCO-2002. Morgan Kaufmann, 2002, pp. 327-334.

[3] Sung-Soon Choi and Byung-Ro Moon. *More Effective Genetic Search for the Sorting Network Problem*. In Proceedings of GECCO-2002. Morgan Kaufmann, 2002, pp. 335-342.

[4] Harrison, M. L., and Foster, J. A. *Co-evolving Faults to Improve the Fault Tolerance of Sorting Networks*. In Proceedings of EuroGP 2004. Springer-Verlag, 2004.

[5] Danny Hillis. *Co-evolving Parasites Improve Simulated Evolution as an Optimization Procedure*. In Proceedings of Artificial Life II (1990). Westview Press, 1991.

[6] Hugues Juillé. Evolution of Non-deterministic Incremental Algorithms as a New Approach for Search in State Spaces. In Proceedings of ICGA-95. Morgan Kaufmann, 1995, pp. 351-358.

[7] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd edition). Addison Wesley, 1998.

[8] Sekanina Lukás. *Evolving Constructors for Infinitely Growing Sorting Networks and Medians*. In SOFSEM: Theory and Practice of Computer Science. Springer, 2004, pp. 314-323.

[9] Marek Piotrów. *Depth Optimal Sorting Networks Resistant to k Passive Faults*. SIAM Journal on Computing, Volume 33, Number 6 (2004), pp. 1484-1512.