Low Auto-correlation Binary Sequences

Hussein Houdrouge & Mikhail Volkhov

February 2019

Solving Optimization Problems with Search Heuristics

MPRI

Contents

1 Introduction **2**
1.1 Problem Definition 2

2 Algorithm Description **2**
2.1 1+1-EA Algorithm 2
2.2 (μ, λ) Algorithm 2
2.3 Steepest Ascent Local Search Algorithm (SALS) 3
2.4 Simulated Annealing (SA) 3
2.5 Tabu Search (TS) 3
2.6 Memetic Algorithms (MA) 4

3 Implementation Details **4**

4 Experimental Results **5**

5 Conclusion **6**

1 Introduction

Low Binary Auto-correlation Sequences (LABS) is a challenging computational problem with many applications in different fields such as communication engineering, mathematics, physics, and computer science [PM16]. For instance, engineers use these sequences as modulation pulses in radar and sonar ranging, and physicists use the optimal solution of the LABS's problem as the ground state of a generalized one dimensional Ising spin system with long-range 4-spin interactions [BBB17]. In this project, we aim to implement and test different optimization techniques and give their statistical comparison. In the following subsection, we will define the problem. In section 2 we will provide a description of the algorithms that we chose to implement and compare. Then we will briefly mention some implementation details. After that, we will show our experimental results.

1.1 Problem Definition

The main objective of the problem is to find a binary sequence S of length N that optimizes a given function. Elements s_i of S are in $\{-1, 1\}$. The auto-correlation of S is denoted by $C_k(S)$ and defined in the following way:

$$C_k(S) = \sum_{i=1}^{N-k} s_i s_{i+k}, \text{ for } k = 0, ..., N-1.$$

The energy of S is defined as:

$$E(S) = \sum_{k=1}^{N-1} C_k(S)^2. \tag{1}$$

Our goal is to find S minimizing E; though it is usually more convenient to rephrase the task as a maximization problem, where we maximize the merit factor

$$F(S) = \frac{N^2}{2E(S)}. \tag{2}$$

In this work, we formulate the task as the maximization of merit factor F. We also view S as a binary vector by implicitly mapping "$false$"s to -1s and "$true$"s to 1.

2 Algorithm Description

In this section, we aim at describing the algorithms implemented in our project. We review several Evolutionary Algorithms (EAs), tabu search (TS) and simulated annealing (SA). Most effective EAs we use combine local and global search heuristics with the classical approaches such as recombination, mutation, etc. All of the algorithms are running in the timeout setting, stopping when the given timeout value is exceeded.

2.1 1+1-EA Algorithm

1+1-EA is a very simple optimization algorithm. It starts by initialising the binary sequence S randomly. Then, the variation phase consists of creating a new sequence y by deriving it from S using standard bit mutation. It is performed in the following way: we choose a random number ℓ from binomial distribution $bin(|S|, 1/|S|)$, then we flip the chosen ℓ bits from S with probability $\frac{1}{2}$. After creating y, we evaluate F on both y and S, and we keep the optimal value. We stop when the timeout is exceeded.

2.2 (μ, λ) Algorithm

The (μ, λ) algorithm is another evolutionary algorithm. It starts with a population of size μ, and uses it to produce a λ children. The child is chosen either by a crossover (we tested a uniform and one-point crossovers), or by the mutation. After the mutation, a selection process is performed – μ offsprings with the highest fitness value are turned into the main population.

2.3 Steepest Ascent Local Search Algorithm (SALS)

The traditional SALS starts from the initial sequence S, then visits all its neighbors by flipping one bit at a time. It replaces the initial sequence by the best neighbor. It continues in the same manner until no improvement is possible. In our timeout-based setting we choose some value S^* randomly and run SALS on it. In the next iteration, we apply the standard bit mutation to it and repeat the procedure. At each iteration, we check if the result of SALS has greater fitness value than our current optimum, and update in this case. The detailed description of the basic SALS is given in [GCF09].

2.4 Simulated Annealing (SA)

SA is an optimization algorithm inspired by the physical process of annealing [BFM18]. It begins with the initial high temperature T and the state S . Given a neighborhood relation, SA generates uniformly at random a neighbor N of S. Then, if N optimizes the merit factor F, we assign N to S. Otherwise, we have the option to accept N, even if it has a lower merit factor, with probability

$$e^{\frac{-(F(S)-F(N))}{T}}.$$

What we just described until now is known as The Metropolis Algorithm as it is explained in [KT06]. SA is just a simple modification to this algorithm which consists of lowering the temperature T at each step according to a given schedule. A detailed discussion of the cooling scheduling can be found in [NA98]. In our implementation, we reviewed two main common cooling schedules. The first one is the linear schedule

$$T(t) = T_0 - \mu t$$

and the second one is the exponential schedule

$$T = T_0 \alpha^t$$

where $\mu, \alpha \in (0, 1)$. we can summarise the SA by the following pseudo-code based on the one in [KT06]. In the experiments, however, we only used the linear cooling, since it showed itself to perform better.

Algorithm 1 SA

```
Start with an initial random solution.
let T ← T_0
for i = 1 to itr do
  let N be chosen uniformly at random from the neighbour of S.
  if F(N) ≥ F(S) then
    S ← N
  else
    with probability e^{-(F(S)-F(N))/T} let S ← N
  end if
  cool the System according to the given Schedule.
end for
```

2.5 Tabu Search (TS)

Tabu search is another local search strategy that can escape local maximums. Our description of the algorithm is based on the [GCF09] up to redefining it as a maximization procedure. The main idea behind the TS is to maintain a tenure table $\mathcal{M}$ of size N, such that we can accept a neighbor only if $\mathcal{M}_i \leq k$ where k is the current iteration. In the beginning, M is empty. The algorithm starts by exploring the neighbors of an initial solution and only accepts the neighbor if it is better than the optimal value or the number of iteration greater than its Tabu value ($\mathcal{M}_i \leq$ Number of Iteration). Algorithm 2 presents the pseudo code of Tabu Search based on the [GCF09].

In our timeout setting we do a slight modification. Instead of iterating $maxIters$ times, we loop until the timeout value is exceeded, and re-randomise the value each $maxIters$ steps.

Algorithm 2 Tabu Search

```
for i = 1 to L do
  M_i ← 0
end for
Initialise currentSequence Randomly
minTabu ← maxIters/10
extraTabu ← maxIrers/50
optSequence ← currentSequence
f_optSequence ← F(optSequence)
for k = 1 to maxIters do
  f+ = −∞
  for i = 1tol do
    neighbour ← flip(currentSequence, i);
    f_neighbour ← F(neighbour)
    if k ≥ M_i or f_neighbour ≥ f_optSequence then
      if f_neighbour ≥ f+ then
        f+ ← f_neighbour; S+ ← neighbour; I+ = i
      end if
    end if
  end for
  currentSequence ← S+
  M_{i+} ← k + minTabu + URand[0, extraTabu)
  if f+ ≥ f_optSequence then
    optSequence ← f+; f_optSequence ← f+
  end if
end for
return optSequence
```

2.6 Memetic Algorithms (MA)

Memetic Algorithm is a meta-heuristics algorithm that makes use of local search algorithms as a subroutine. Our implementation is based on the [GCF09]. The main idea behind the MA is to apply evolutionary strategies on the vectors in our population, while the actual optimization is performed by the local search algorithm. First, we create a random population with a given size. Then we apply some recombination techniques with a probability p_x to produce the offspring population from the parents with the option to mutate the offspring with probability p_m. An example of a recombination technique could be a uniform crossover, single point crossover, or K-points crossover (we use a single point crossover). At this stage, the algorithm runs a local search algorithm such as Tabu Search on the offsprings. After optimizing all the offsprings, the algorithm replaces the main population with the offspring one. The algorithm repeats the same process until it reaches the stopping condition (timeout limit in our case). A more detailed description is mentioned in [GCF09].

3 Implementation Details

We implemented all the above-motioned algorithms in $C++$. We used also *python* to perform the data analysis. One can access the detailed code of our implementation in the GitHub repository[1]. To accelerate some features of the algorithms, we modified the implementation of some function such as standard bit mutation as it is described in section 2.1. Moreover, we used a different algorithm to accelerate the evaluation

[1] https://github.com/CHoudrouge4/osh

of the merit factor of a neighbor sequence in TS and SALS. To compute the merit factor after one flip, we used the algorithm described in [GCF09]. This algorithm reduces the complexity of computing the merit factor from $O(N^2)$ to $O(N)$, and it goes as follows. First, it creates a $(N-1) \times (N-1)$ table $\mathcal{T}(S)$ such that $\mathcal{T}(S)_{ij} = s_j s_{i+j}$ for $j \leq N-i$. Then, it creates an array $\mathcal{C}(S)$ of length $N-1$ such that $\mathcal{C}(S)_k = C_k(S)$ for $k \leq N-1$. We can notice that flipping one bit in S multiplies the cell where the bit is involved by -1. Therefore, to compute the new value, it is enough to take the right C_k and subtract 2 times the cells where the flipped bit is involved. For more details consider the following pseudo-code.

Algorithm 3 FlipValue(S, i, $\mathcal{T}$, $\mathcal{C}$)

```
E ← 0
for p = 1 to L − 1 do
  v ← C_p
  if p ≤ L − i then
    v ← v − 2T_pi
  end if
  if p ≤ i then
    v ← v − 2T_{i−p}
  end if
  E ← E + v^2
end for
return N^2/(2E)
```

Implementation of all other algorithms are not heavily optimized and follow the pseudocode, with the exception of obvious language-dependent optimizations and reducing the data accesses.

4 Experimental Results

Solvers and parameters Our implementation of 1+1-EA and (μ, λ) were too slow to get any results we can argue about, so we excluded them from the practical consideration. We have collected and analysed the data for the SA, SALS, TS and MA (over TS) algorithms. The parameter choice of SA was motivated by the simple iterative grid search in the parameter space, and the final values we use are: linear cooling with $\mu = 0.15$ and $t_0 = 15000$; We suspect that these parameters give solver more freedom to escape the local optima, which might lead to the best observed average *runtime*. Parameters for TS are choosed almost as in [GCF09]: we choose the *maxIter* randomly in the range $[N - N/4, N + N/4]$, and set $minTabu = maxIter/10, extraTabu = maxIter/50$. For the MA (over the TS), we set the population size to 130, use $px = 0.9$ and $pm = 1/N$ – these parameters again follow the [GCF09], but the population size is chosen slightly higher than the suggested 100 because of the empirical evidence of the former leading to the better performance.

Experiments setup All our experiments were run on the 8 core Intel Xeon at 3GHz (Amazon AWS c5.2xlarge instance). We have run all the experiments on all 8 cores, considering the performance decrease due to the full CPU use negligible. The measurement techniques we used are based on the strategies developed in [BBB17] and [GCF09]. As in [BBB17], we collected both the *runtime* values that follow the execution and *cntProbe* – the number of calls to the LABS instance. But in our particular implementation, the average correlation between these variables (for the particular runner over the range of instance sizes) is sometimes as low as 0.81, so we decided not to include this parameter into the analysis and only operate with the raw runtime. We call *hits* the number of times the solver achieved the optimum value within the time limit. The *hitRatio* is the number of hits over the number of runs. All the runs were performed with a fixed timeout of 90 seconds averaged on 50 runs, and we considered the values of N from 15 to 55 for all the algorithms except the SA, which was tested up to $N = 40$. All the code used for the analysis is available in the repository as a single ipython notebook.

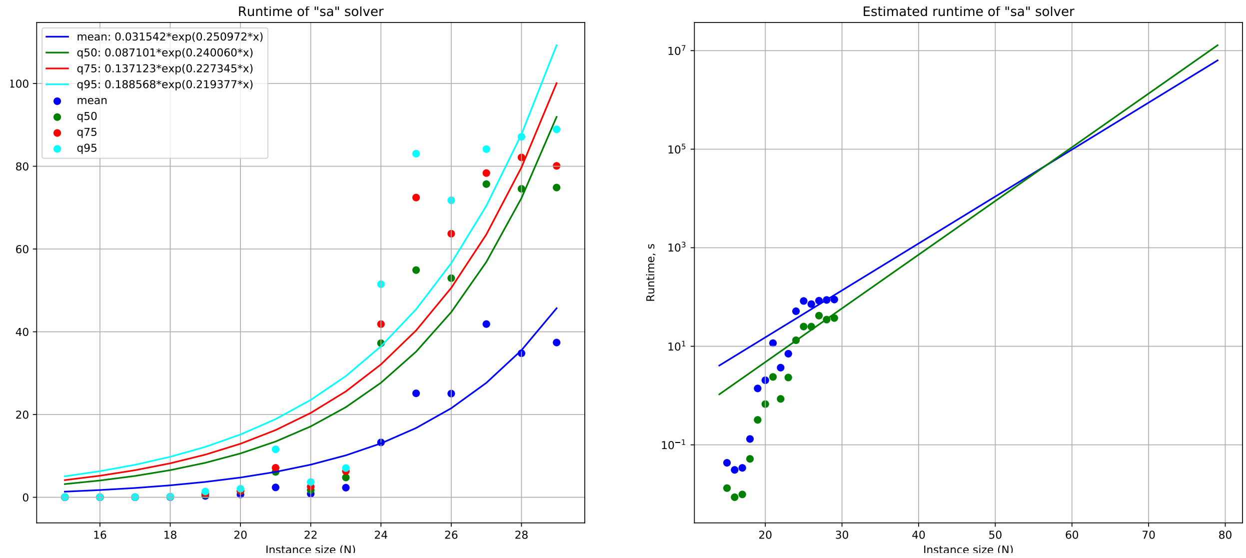


Figure 1: Runtime analysis of the SA solver

Results analysis First, we analyze the asymptotic runtime of the algorithms within the range of N that leads to the *hitsRatio* close to one. For each N, the distribution of *runtime* is close to the exponential, and we would like to know how the mean and percentiles behave with the growth of N. Figures 1 to 4 illustrate the asymptotic performance of the solvers. SA performance is the lowest among the solvers reviewed, and we start to exhibit the low hit ratio already at the low N, other solvers start to slow down at bigger values. We perform the exponential curve fitting to the data to predict the runtime for the bigger N (as it is done in [BBB17], using the mean runtime value and the three quantiles. The second plot of these figures shows the asymptotic behaviour of our predicted model.

Next, in the Figure 5, we compare all the asymptotic runtime models that we have built, observing the bad asymptotic behaviour of SA and overall better runtime of the other three solvers, with TS leading.

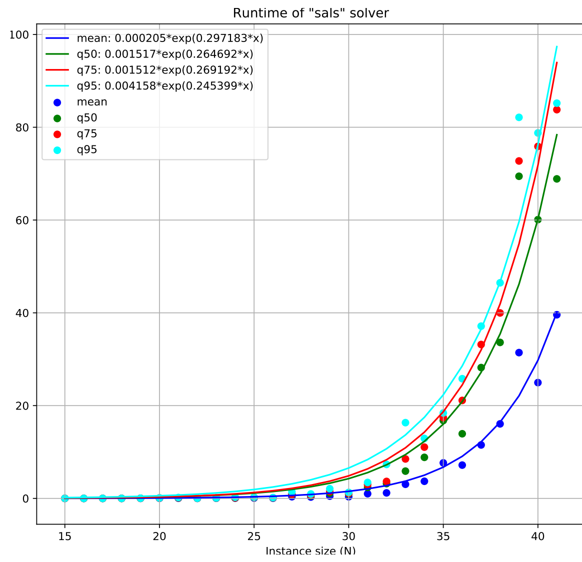
Figures 6-9 show how the runtime value is distributed for the given N. Each chart has its asymptotic model added, for the 95th percentile (the value one can choose, for instance, for the timeout) and the mean.

Figure 10 represents how far are the solution we have obtained from the global optimum. The interval of instance size is selected such that for less N we get the $hitRatio = 1$.

Overall, we see that the performance of the solvers decrease in the following sequence: TS, MA, SALS, SA.

5 Conclusion

We have implemented, optimized and tested a number of different algorithms. The results obtained show among the four algorithms tested, the SA has the lowest convergence speed, while TS is the fastest one. We could not reproduce the analysis of [GCF09] showing the strictly better general performance of MA over the TS (in our implementation, MA is slightly slower than TS), which may be due to the implementation details slowing down the MA.



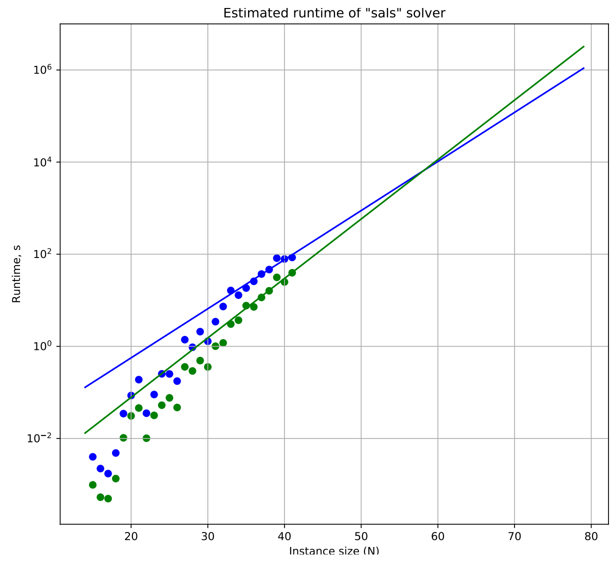
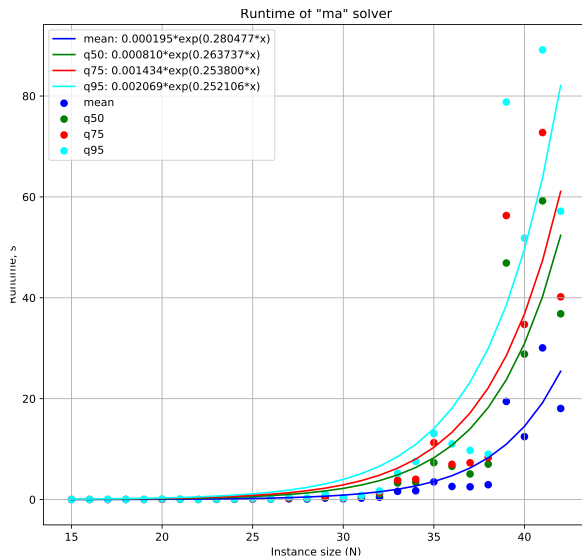


Figure 2: Runtime analysis of the SALS solver



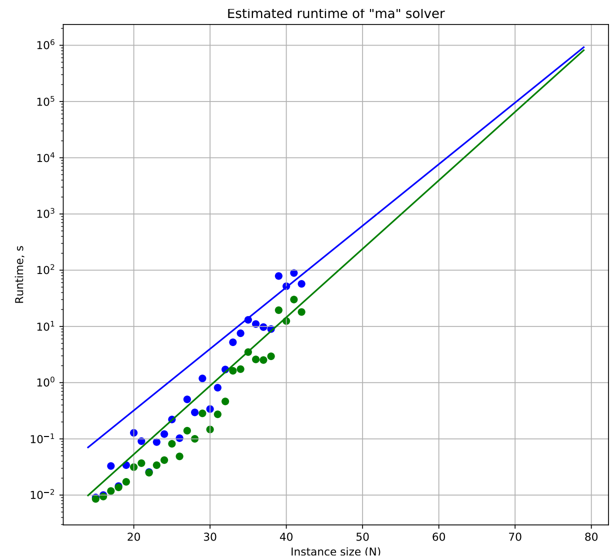


Figure 3: Runtime analysis of the MA solver

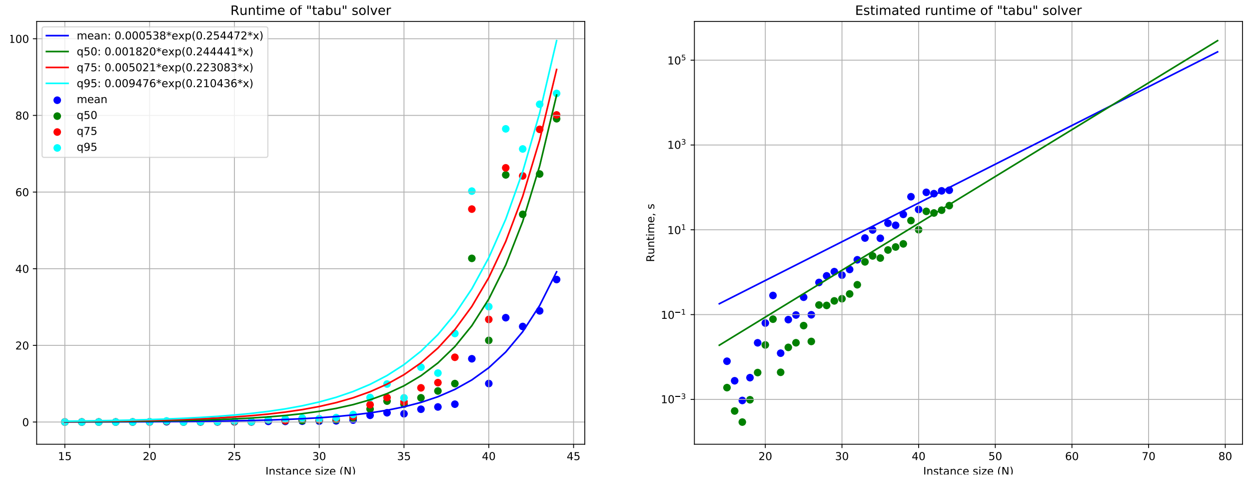


Figure 4: Runtime analysis of the tabu solver

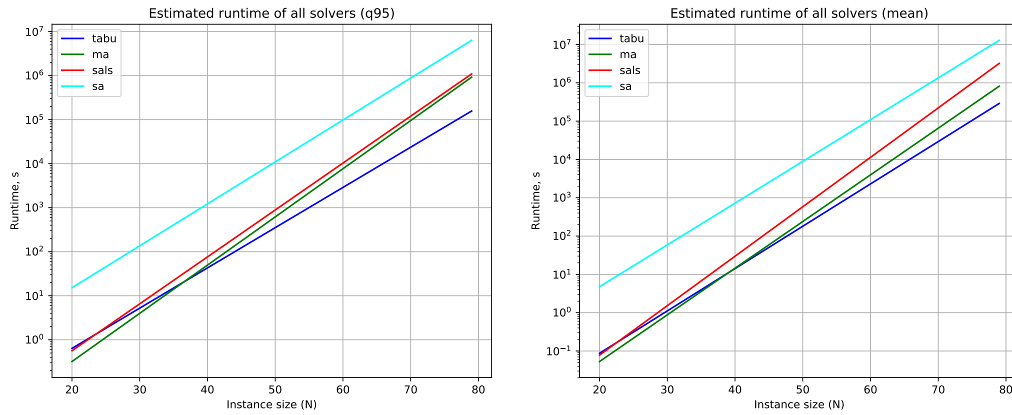


Figure 5: All runtime models comparison

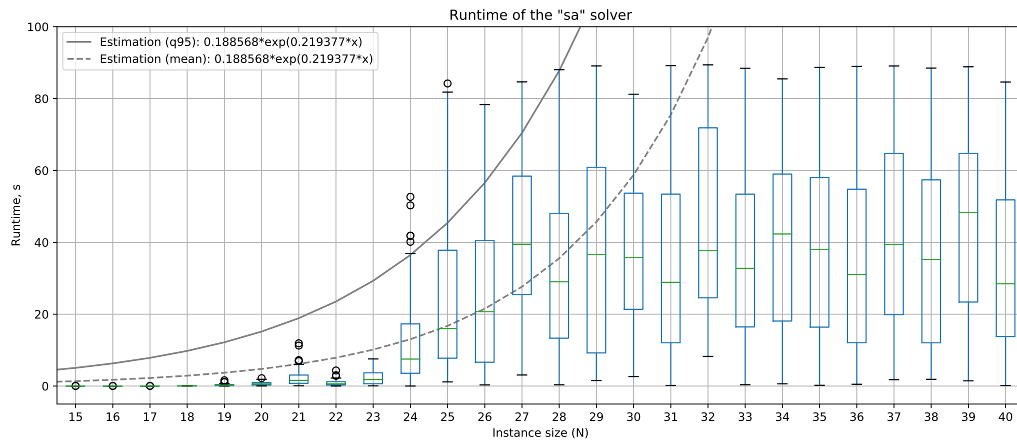


Figure 6: Runtime performance of the SA solver

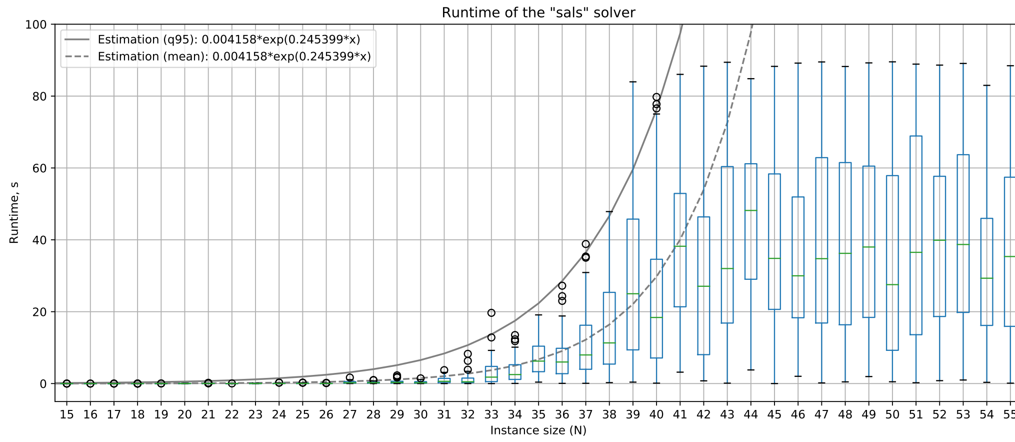


Figure 7: Runtime performance of the SALS solver

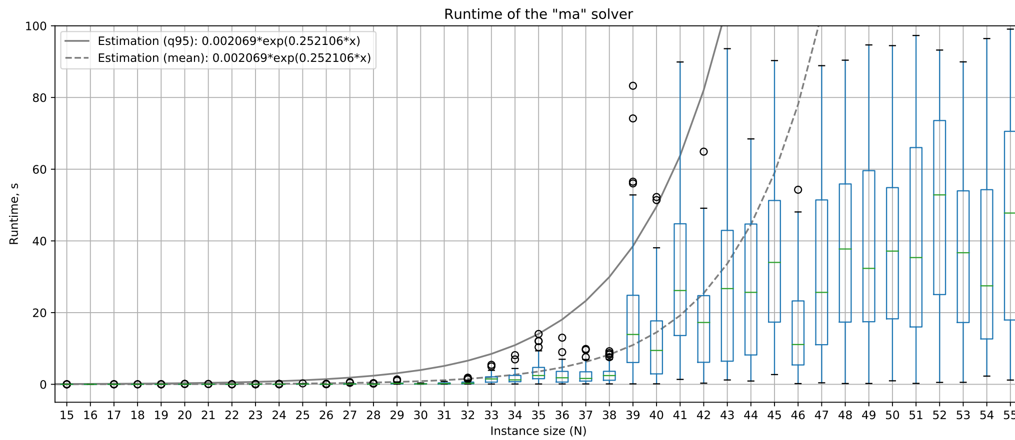


Figure 8: Runtime performance of the MA solver

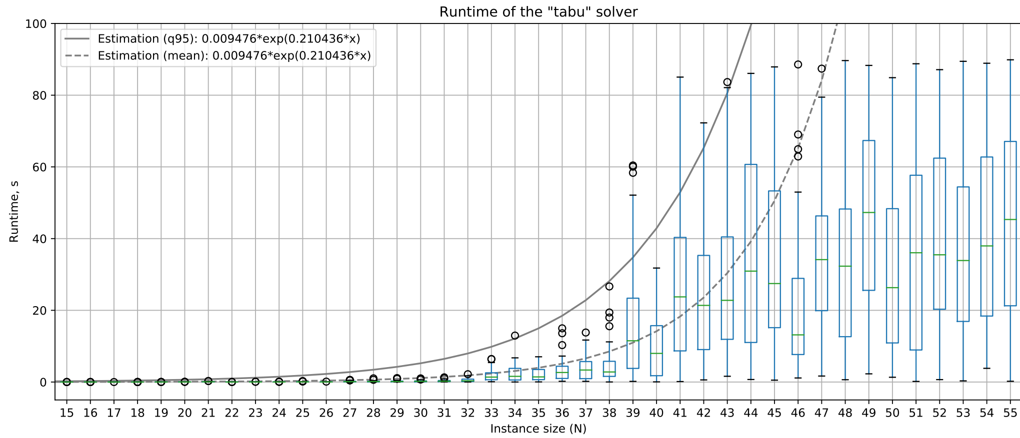


Figure 9: Runtime performance of the tabu solver

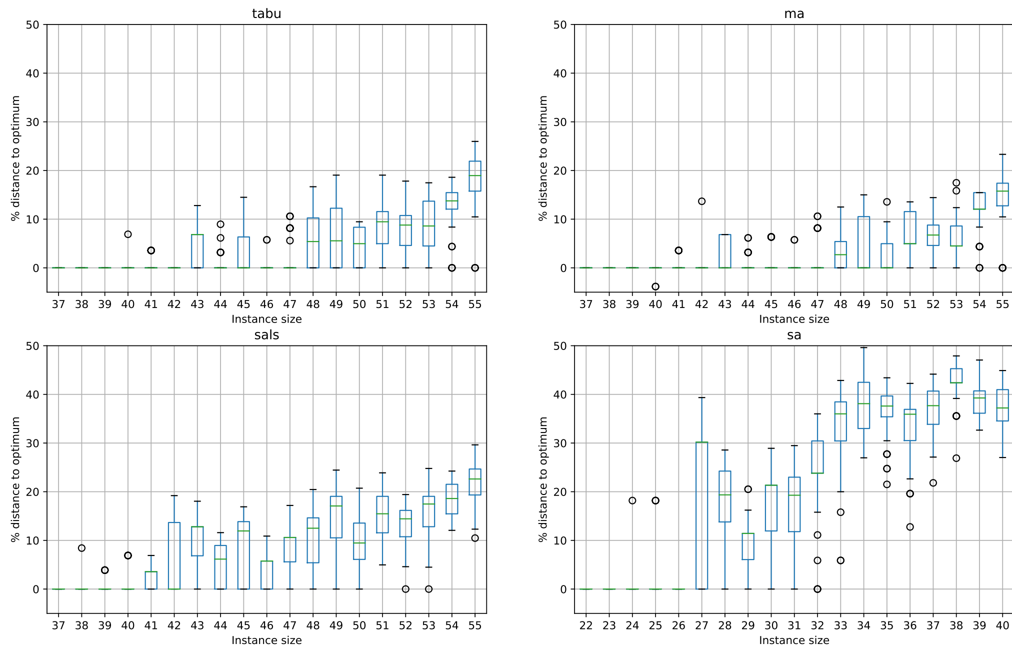


Figure 10: Relative distance to the optimum. The hit ratio of SA starts to decrease at $N = 26$, while other algorithms start to experience low hit ratios only at $N > 40$.

References

[BBB17] Borko Bošković, Franc Brglez, and Janez Brest. Low-autocorrelation binary sequences: On improved merit factors and runtime predictions to achieve them. *Applied Soft Computing*, 56:262 – 285, 2017.

[BFM18] T. Baeck, D.B. Fogel, and Z. Michalewicz. *Evolutionary Computation 1: Basic Algorithms and Operators*. CRC Press, 2018.

[GCF09] José E. Gallardo, Carlos Cotta, and Antonio J. Fernández. Finding low autocorrelation binary sequences with memetic algorithms. *Applied Soft Computing*, 9(4):1252 – 1262, 2009.

[KT06] J. Kleinberg and É. Tardos. *Algorithm Design*. Alternative Etext Formats. Pearson/Addison-Wesley, 2006.

[MZB98] Burkhard Militzer, Michele Zamparelli, and Dieter Beule. Evolutionary search for low autocorrelated binary sequences, 1998.

[NA98] Yaghout Nourani and Bjarne Andresen. A comparison of simulated annealing cooling strategies. *Journal of Physics A: Mathematical and General*, 31(41):8373–8385, oct 1998.

[PM16] Tom Packebusch and Stephan Mertens. Low autocorrelation binary sequences. *Journal of Physics A: Mathematical and Theoretical*, 49(16):165001, mar 2016.