Report of the Project: Design, Implementation, and Testing of Heuristics for the Low Autocorrelation Binary Sequence (LABS) problem

Narges Alavi Samani

Abstract—In this project, the aim is to implement, test, and compare heuristic search algorithms for Low Autocorrelation Binary Sequence (LABS). The goal in LABS problem is to maximize the merit, which has various applications in different areas of science. In this project, two (1+1)EA and simulated annealing algorithms are implemented in C++. Although there are more criteria which should be considered, these algorithms are compared together with respect to the merit they found and the time they consumed. The results show that (1+1)EA obtains better results, in the implementations of this project; while simulated annealing consumed less time.

Index Terms—Low Autocorrelation Binary Sequence, heuristic search, evolutionary algorithms, simulated annealing.

1 Introduction

To formulate the aperiodic Low-autocorrelation Binary Sequence (LABS), consider a binary sequence of length N, $S = s_1 s_2 ... s_N$, $s_i \in \{+1, -1\}$. The autocorrelation function is

$$C_k(S) = \sum_{i=1}^{N-k} s_i s_{i+k} \tag{1}$$

And, the target is to minimize the following energy function:

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

or equivalently, to maximize the merit factor F [1]:

$$F(S) = \frac{N^2}{2E(S)}. \tag{3}$$

Finding a binary sequence with the best merit has significant applications in various areas, such as communication engineering, mathematics, physics, chemistry, cryptography, and etc. Thus, solving this problem is important and useful in these different areas of science [2], [3].

Achieving the optimum sequence solution is notably harder than the special cases of LABS application in the mentioned areas. According to [4], the best proven optimal merit factors, obtained by the exhaustive search, for the formulation of LABS in equation 3 are currently known for $N \leq 60$ only. In [5] – which is a restored version of the web page of LABS best merit factors and solutions, up to the sequence length of L = 304, compiled by Joshua Knauer in 2002–, there are tables of updates on the best known figures of merit, as well as, the number of unique solutions in canonic form and the solutions themselves.

Using arguments from statistical mechanics [6], the asymptotic value for the maximum merti factor F is introduced in [7]:

$$\text{as} \quad L \to \infty, \quad \text{then} \quad F \to 12.3248 \tag{4}$$

The challenge of finding a solution to converge to this value as the length of the sequence increases is still in progress.

2 Algorithms and Implementation

According to the current state of our knowledge, the only way to find the optimum solution of LABS is to use exhaustive search. However, due to the search space of this method which grows exponentially, 2^N, this approach is limited for the small value of N. Thus, this method is not suitable in general, and for large N.

Heuristic techniques are designed to solve the problems where classical methods either fail to find an exact solution or execute too slowly to find any. In heuristics techniques, there are trade-offs between optimality, completeness, accuracy, precision, or speed [8].

One of the groups of heuristic algorithms for optimization is Evolutionary Algorithm (EA). Evolutionary algorithms are general randomized search heuristics, inspired by the concept of natural evolution [9]. Figure 1 perfectly describe the outline of a generic evolutionary algorithm.

2.1 (1+1)EA

The first algorithm, implemented in this project in C++, is (1+1)EA, which is a special case of $(\mu + \lambda)$EA. Assume that $f(x)$ is the function we want to maximize by using (1+1)EA, algorithm 1 shows the procedure.

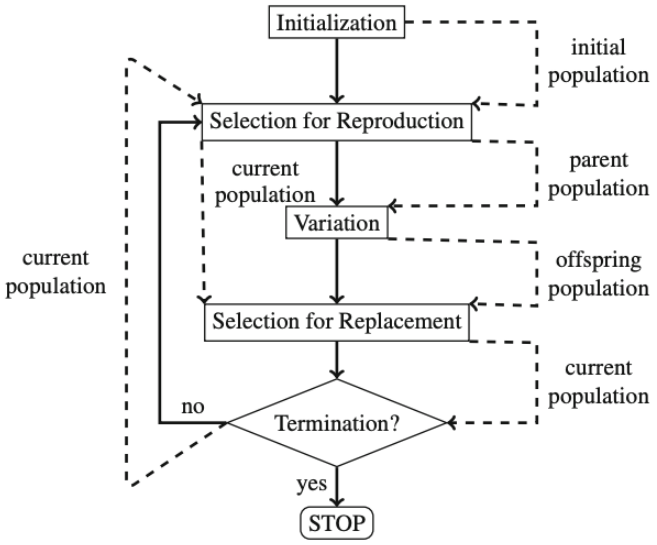

Fig. 1. Outline of a generic evolutionary algorithm. Arrows show control flow, dashed arrows show data flow. the figure is from [9]

Algorithm 1 (1+1)EA

1: **Initialisation**
Choose $x_0 \in \{0,1\}^n$ *uniformly at random.*
$t := 0$
2: **Variation**
Create y *by flipping each bit of* x_t *with* $p_m = 1/n$.
3: **Selection for Replacement**
4: **if** $f(y) \geq f(x_t)$ **then**
5: $x_{t+1} := y$
6: **else**
$x_{t+1} := x_t$
7: **end if**
8: $t := t + 1$, *Continue at line 2*

There are two different kinds of performance criteria we should take into account. First, fixed-target perspective, $\mathbf{T}((1+1)EA, F(S), v)$, which is the number of queries that $(1+1)EA$ needs to find a solution sequence S for equation 3 with $F(S) \geq v$. Second, fixed-budget perspective, $\mathbf{V}((1+1)EA, F(S), t)$, which is the quality of best-so-far solution that $(1+1)EA$ has found with its first t steps. Assuming that we have no idea about the value of optimums, fixed-budget perspective is considered; and, a fixed number of iteration is considered as a termination criterion. In this implementation, due to the not too big value of 2^N for $N \leq 20$, in this range, the number of iterations is 2^{N-1}, which is the half needed in an exhaustive search but in my results it was always enough to find the maximum in this range. For $20 < N \leq 45$, number of iterations (queries to $F(s)$) is $\sum_{n=1}^{N} nlog_2(n)$. And for the each of the rest of $45 \leq N \leq 143$, the program execute for 1 million iteration. (To achieve better results, it would be better to repeat the algorithm several times and then report the expected value of results; but since it is too time-consuming, this work was avoided)

For Initializtion step, `std::random_device` , `std::mt19937`, and `std::uniform_real_distribution` in `<random>` libarary is used to generate a uniform random sequence [10]. `std::random_device` is a uniformly-distributed integer random number generator that produces non-deterministic random numbers; `std::mt19937` is Mersenne Twister pseudo-random generator of 32-bit numbers with a state size of 19937 bits; `std::uniform_real_distribution` produces random floating-point values i, uniformly distributed on the interval $[a, b)$, that is, distributed according to the probability density function $p(i|a, b) = \frac{1}{a-b}$. For each s_i, `std::uniform_real_distribution` gives a real number between 1 and 2, uniformly at random; for random number less than 1.5 x_i is -1, and it is $+1$ otherwise.

2.2 Simulated Anealing Algorithm

The other heuristic algorithm, implemented in this project, is known as simulated annealing [11]. The mutation operator, for this algorithm, flips at most a single bit. As it is clear in Algorithm 2, simulated annealing search through the neighbours. In addition, to avoid getting stuck in local optima, replacement can happen with probability $min\{1, e^{(f_{new\ value} - f_{current\ value})/T(t)}\}$ despite $f_{new\ value} < f_{current\ value}$. As it is obvious, this probability depends on the difference of the value of the function, we want to maximize, of the current sequence and of the sequence created in Variation step. Additionally, the probability depends on function $T(t)$, named temperature function. Temperature function is monotonically decreasing [11].

Algorithm 2 Simulated Annealing

1: **Initialisation**
Choose $x_0 \in \{0,1\}^n$ *uniformly at random and* $opt = f(x_0)$.
$t := 0$
2: **Variation**
Choose $i \in [N], N = 2^n$, *uniformly at random.*
Create $y = x_t$, *and flip* y_i.
3: **Selection for Replacement**
4: **if** $f(y) > f(x_t)$ **then**
5: set $x_{t+1} = y$
6: **if** $opt < f(y)$ **then**
7: set $opt = f(y)$
8: **end if**
9: **end if**
10: With probability $min\{1, e^{(f(y)-f(x_t))/T(t)}\}$ set $x_{t+1} = y$.
11: $t := t + 1$, *Continue at line 2*

To implement simulated annealing, Initialization step is the same as (1+1)EA. For generating uniform random number, I used the same functions from `<random>` library but replacing `std::uniform_int_distribution` [10] with `std::uniform_real_distribution` as we want to find an index, which is integer, uniformly at random. For choosing the index of the bit ,which is going to be flipped, `std::uniform_int_distribution` in `<random>` library is used. By giving 1 and N, the beginning and the end of the desired interval, as the input,`std::uniform_int_distribution`, produces random integer values, uniformly distributed on the closed interval $[1, N]$. In this project, I use different temperature

functions for $T(t)$, namely $T(t) = 1/t$ and $T(t) = 1/lnt$, to see how different their results will be.

I repeat (1+1)EA, simulated annealing with $T(t) = 1/t$, and simulated annealing with $T(t) = 1/lnt$ for 2 million number of iterations. In addition, all of the algorithms ran for 20 times and the results are expected values.

3 Results

In this project, MATLAB is used to visualized the results. Figure 2 shows the results achieved from (1+1)EA implementation. Figure

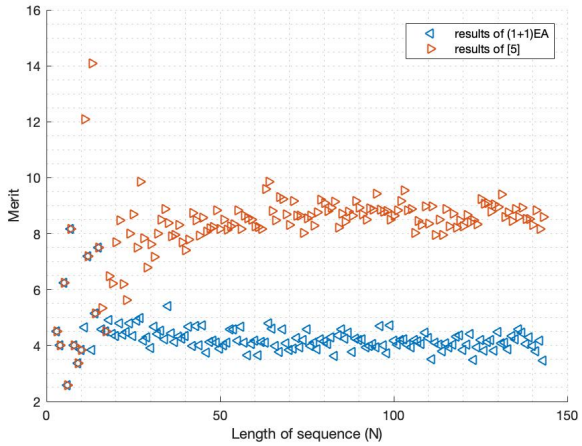

Fig. 2. Comparisons between the results in [5] and (1+1)EA. For this results, The number of queries to F(s) varies in ranges. for $N \leq 20$ the number of iterations is 2^{N-1}, for $20 < N \leq 45$, it is $\sum_{n=1}^{N} nlog_2(n)$; and for the each of the rest of $45 \leq N \leq 143$, there are 1 million iterations.

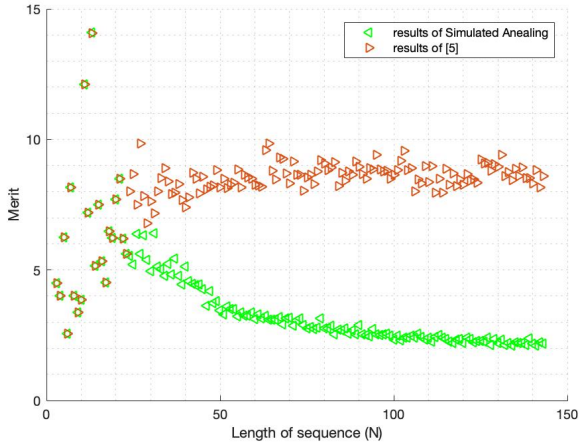

Fig. 3. Comparisons between the results in [5] and simulated annealing. For this results, the same as results of (1+1)EA in Figure 2, The number of queries to F(s) varies in ranges. for $N \leq 20$ the number of iterations is 2^{N-1}, for $20 < N \leq 45$, it is $\sum_{n=1}^{N} nlog_2(n)$; and for the each of the rest of $45 \leq N \leq 143$, there are 1 million iterations. In this simulation, tempreture function is $T(t) = 1/t$.

In this project, `clock()` function is used to measure CPU time; and converted it to ms by dividing the difference of the output of the clock, before and after calling the algorithm function, into `CLOCKS_PER_SEC`/1000.

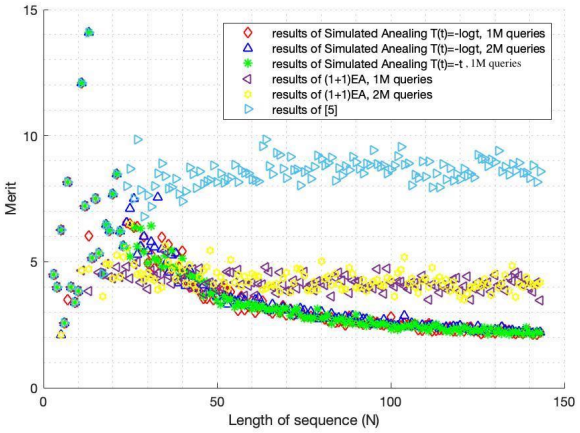

Fig. 4. Comparisons between the results in [5] and simulated annealing. For this results, the same as before, for $N \leq 20$ the number of iterations is 2^{N-1}, for $20 < N \leq 45$, it is ; and for the each of the rest of $45 \leq N \leq 143$, it is mentioned in the picture and varies between 1 million and 2 million.

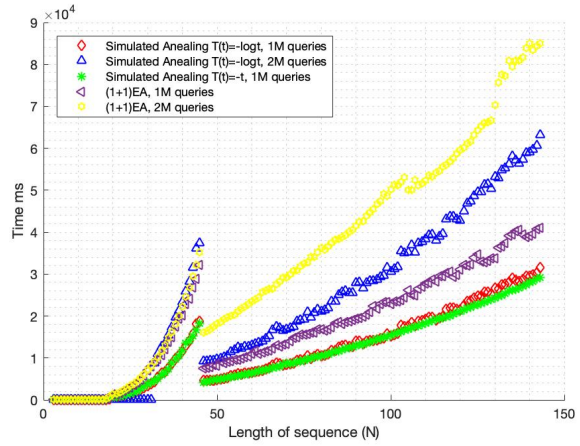

Fig. 5. This figure shows the average time, spent on executing the algorithms.

4 Discussion

In general, it is very difficult, and maybe impossible, to have a universal, accurate, and fair comparison of two randomized search heuristics. According to the result section, (1+1)EAs generally perform better than simulated annealing. One of the reasons of such better performance is that (1+1)EAs can reach any points in the search space in only one step; while, simulated annealing has to make many steps and may have to accept many decreasing steps in order to do so. In addition, the choices of appropriate cooling schedule, namely temperature function and initial temperatures, significantly affect the performance of simulated annealing. In the implementation of simulated annealing, done in this project, only the matter of being monotonically decreasing is considered; however, there are some cases where increasing temperature can be advantageous [12].

According to Figure 4, for both simulated annealing and (1+1)EA, the differences of 1 million iterations with 2 million ones is not significant, and for higher amounts of N, these differences become lower and lower. One of the reasons is

that, the search space increases exponentially by N; thus, just doubling the number of iterations cannot be that much effective.

In these implementations, two temperature functions of $T(t) = 1/t$ and $T(t) = 1/lnt$ does not make huge differences in the results. In addition, it is good to notice that T_0 in these implementations is 0. While, changing the value of T_0 can affect the results obtained from simulated annealing.

There is no special reason for the number of iterations chosen in this project for simulation of algorithms for $20 < N \leq 45$ but to have dependency with N in addition to having a higher order of complexity than $O(nlogn)$ which is more promising for linear functions, not LABS.

Figure 5 shows the CPU time in ms. For $N \leq 20$, there are almost no differences, due to having a small search space. For $20 < N \leq 45$, consumed time, as well as the differences between algorithms, grows comparably faster. Although dependency of the number of iterations to the value of N is more reasonable, and according to results, it can bring about better outcomes, fast growth of consumed time causes to decide to have a constant number of iterations. For $45 < N$, clearly there is almost no differences between simulated annealing with $T(t) = 1/t$ and $T(t) = 1/lnt$. Consumed time gets twice for both (1+1) and simulated annealing by doubling the number of iterations from 1 million to 2 million. Due to the variation level of (1+1)EA, which causes having a loop from 1 to N in implementation, in each iteration, consumed time for (1+1)EA grows faster than simulated annealing.

5 Conclusion / Future work

In this project, by using implementation, a result-based comparison between (1+1)EA and simulated annealing for LABS problem is presented. By means of the implementations, one learns that in spite of similarities, the performance of (1+1)EA and Simulated annealing can be different. Although one cannot have a precise comparison for heuristic search algorithms in general, the results in this project, noticing to the parameters chosen and the number of iterations, shows that (1+1)EA has better results than simulated annealing, and also its consumed time is more than simulated annealing. There is always a trade-off between consumed time and accuracy of the results; in this project, this trade-off is clear too.

To have a better comparison, one can test more different temperature functions and initial temperature for simulated annealing, as well as, other evolutionary algorithms.

References

[1] B. Boskovic, F. Brglez, and J. Brest, "Low-autocorrelation binary sequences: on the performance of memetic-tabu and self-avoiding walk solvers," *CoRR*, vol. abs/1406.5301, 2014.

[2] E. Marinari, G. Parisi, and F. Ritort, "Replica field theory for deterministic models: I. binary sequences with low autocorrelation," *Journal of Physics A: Mathematical and General*, vol. 27, pp. 7615–7645, dec 1994.

[3] J. Jedwab, "A survey of the merit factor problem for binary sequences," in *Sequences and Their Applications - SETA 2004* (T. Helleseth, D. Sarwate, H.-Y. Song, and K. Yang, eds.), (Berlin, Heidelberg), pp. 30–55, Springer Berlin Heidelberg, 2005.

[4] S. Mertens, "Exhaustive search for low-autocorrelation binary sequences," *Journal of Physics A: Mathematical and General*, vol. 29, pp. L473–L481, sep 1996.

[5] B. Bošković, F. Brglez, and J. Brest, "A GitHub Archive for Solvers and Solutions of the `labs` problem." For updates, see https://github.com/borkob/git$_labs.$, $January$2016.

[6] Bernasconi, J., "Low autocorrelation binary sequences : statistical mechanics and configuration space analysis," *J. Phys. France*, vol. 48, no. 4, pp. 559–567, 1987.

[7] M. Golay, "The merit factor of long low autocorrelation binary sequences (corresp.)," *IEEE Transactions on Information Theory*, vol. 28, pp. 543–549, May 1982.

[8] J. Pearl, "Heuristics: Intelligent search strategies for computer problem solving," 1984.

[9] T. Jansen, *Analyzing Evolutionary Algorithms: The Computer Science Perspective*. Springer Publishing Company, Incorporated, 2013.

[10] "¡random¿ library in c++11." http://www.cplusplus.com/reference/random/. Accessed: 2019-02-9.

[11] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing.," *Science*, vol. 220 4598, pp. 671–80, 1983.

[12] S. Droste, T. Jansen, and I. Wegener, "Dynamic parameter control in simple evolutionary algorithms," in *FOGA*, 2000.