Parameter-less Late Acceptance Hill-Climbing

Mosab Bazargani
Operational Research Group
School of Electronic Engineering and Computer Science
Queen Mary University of London
London, United Kingdom E1 4FZ
m.bazargani@qmul.ac.uk

Fernando G. Lobo
DEEI-FCT & CENSE
Universidade do Algarve
Campus de Gambelas
Faro, Portugal 8005-139
fernando.lobo@gmail.com

ABSTRACT

The Late Acceptance Hill-Climbing (LAHC) algorithm has been recently introduced by Burke and Bykov. It is a simple, general purpose, one-point search metaheuristic that has similarities with Simulated Annealing (SA) in the sense that worsening moves on a current solution can be accepted. One of its advantages relative to Simulated Annealing is that no cooling schedule is required and its sole parameter, the so-called *history length*, has a more meaningful interpretation from the application point of view and is therefore easier to specify by a user. In this paper we show that even this single parameter can be eliminated, making LAHC simpler to apply in practice. The validity of the method is shown with computational experiments on a number of instances of the Travelling Salesman Problem.

CCS CONCEPTS

•Computing methodologies → Optimization algorithms; Randomized search;

KEYWORDS

Late Acceptance Hill-Climbing, parameter-less search algorithms, local search, metaheuristics

ACM Reference format:
Mosab Bazargani and Fernando G. Lobo. 2017. Parameter-less Late Acceptance Hill-Climbing. In *Proceedings of GECCO '17, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3071178.3071225

1 INTRODUCTION

One-point randomized search algorithms, often also called *local search* algorithms, are among the most popular optimisation methods used in practice. One of their major advantages is their simplicity of use and ease of implementation.

This paper addresses a specific algorithm of this family, the Late Acceptance Hill-Climbing (LAHC), which has been introduced by Burke and Bykov [3, 4] and shown to be competitive (and often superior) to other algorithms of its kind, such as Simulating Annealing (SA) [12], Threshold Accepting (TA) [7], and the Great Deluge Algorithm (GDA) [6], when applied to Travelling Salesman and Exam Timetabling benchmark problems. LAHC has also been successful in optimisation competitions. In 2011, it was the winner of the International Optimisation Competition (IOC) organized by SolveIT Software Pty Ltd, an Australian based company co-founded by Zbigniew Michalewicz, a well-known researcher in the Evolutionary Computation (EC) field. In that competition the goal was to develop a Java command-line application to solve the largest constrained Magic Square problem within one minute of run time. LAHC has also been incorporated in real-world software systems, an example being the constraint satisfaction solver OptaPlanner, an open source project by Red Hat (http://www.optaplanner.org/).

One of the major advantages of LAHC compared with other algorithms of its kind, such as SA, TA, and GDA, is its simplicity of use. LAHC has a single parameter, the *history length*, whose meaning appears to be well understood and directly related to runtime execution and solution quality. Because of that, Burke and Bykov argue that it is simpler to use compared with the aforementioned algorithms. In this paper we go a step further and simplify even more the application of LAHC by eliminating that sole parameter. We do so with a technique that has been successfully used in automating the population size parameter in a variety of Evolutionary Algorithms (EAs).

The next section presents background material needed to understand the paper. It starts with a detailed description of the LAHC algorithm and highlights its advantages compared with SA and other related algorithms. A discussion of the effect of the history length parameter on the search is presented and a connection is made to the related problem of population sizing in EAs. Building on that, Section 3 presents a parameter-less version of the LAHC algorithm. The validity of the method is shown with a number of computational experiments applying it to Travelling Salesman Problem (TSP) benchmark instances that were also used in the work of Burke and Bykov [4]. Section 4 presents a refinement of the technique proposed in Section 3 with the purpose of speeding up the search for good solutions. Again, computational experiments validate the method. The paper ends with a brief summary and major conclusions of the work.

2 BACKGROUND

One-point randomized search algorithms are iterative procedures that maintain a *current solution* which is perturbed until a specified stopping criterion is reached. At each iteration, the current solution is perturbed yielding a candidate solution which can be accepted, becoming the new current solution for the next iteration, or rejected, in which case the current solution remains unchanged.

The perturbation, a *mutation* in the parlance of Evolutionary Computation, is usually achieved by a special purpose operator (or set of operators) for the problem at hand.

One of the most well known algorithms of this class is the stochastic hillclimber, also known as the (1+1) Evolutionary Algorithm, where the candidate (mutated) solution replaces the current one if and only if its quality is better or equal than that of the current solution.

Only accepting better or equal moves may be too restrictive for a proper exploration of the search space. Indeed, it has been recognized that it is beneficial to accept worsening moves sometimes as they allow the possibility of escaping local optima more easily. Several algorithms allow precisely such worsening moves. Among them, the most well-known is probably Simulated Annealing (SA) [12]. In SA, worsening moves are accepted with probability $P = exp((C - C^*)/T)$, with C and C^* denoting the cost function value of the current and candidate solution (here and throughout the paper we assume a minimization problem), respectively, and T is a control parameter, the so-called *temperature*, which has to be set with an initial value and varies through time as the search proceeds according to a *cooling schedule* that needs to be specified.

Like SA, other related algorithms such as TA and GDA, also require a control parameter that is regulated to achieve some sort of cooling schedule, where the probability of accepting a worsening move tends to decay through time, and in effect approaches zero as time goes by. Unfortunately, the optimal form of setting a cooling schedule is problem/instance-dependent, and although there are empirical recommendations, in practice it requires manual tuning to be effective.

Algorithm 1: Late Acceptance Hill-Climbing (LAHC).

```
Input  : The history length, $L_h$.
Output : A solution to a given problem.
1  Produce an initial solution $s$                // Usually at random
2  Calculate its cost function value $C(s)$
3  forall the $k \in \{0 \dots L_h-1\}$ do
4      $f_k = C(s)$
5  $I = 0$                                        // Iteration counter
6  $I_{idle} = 0$                                 // Idle iteration counter
7  do until ($I > 100000$) and ($I_{idle} > I \times .02$)
8      Construct a candidate solution $s^*$
9      Calculate its cost function value $C(s^*)$
10     if $C(s^*) \geq C(s)$ then
11         $I_{idle} = I_{idle} + 1$
12     else
13         $I_{idle} = 0$                         // Reset counter
14     $v = I$ mod $L_h$                          // Virtual beginning
15     if $C(s^*) < f_v$ or $C(s^*) \leq C(s)$ then
16         $s = s^*$                              // Accept candidate
17     else
18         $s = s$                                // Reject candidate
19     if $C(s) < f_v$ then
20         $f_v = C(s)$                           // Update the fitness array
21     $I = I + 1$;
22 return $s$
```

2.1 Late Acceptance Hill-Climbing

Similarly to these, the LAHC algorithm can also accept worsening moves. But unlike SA, TA, and GDA, it does not require a cooling schedule. Instead, the algorithm uses a list to *memorize* previous values of the current solution's quality. The size of the list is called the *history length* L_h, and is the sole parameter of the algorithm. The basic idea of LAHC is that a candidate solution is compared with a solution which was current several iterations before; more precisely L_h iterations before. This contrasts with a traditional hillclimber or a (1+1)-EA where the candidate solution is compared with the current solution of the immediate previous iteration. It is straightforward to see that LAHC degenerates into a (1+1)-EA by setting $L_h = 1$.

Note that the list contains the cost function (fitness) values only, not the solutions themselves. Again, this contrasts with other search algorithms that memorize previously visited solutions, such as Tabu Search [8] and all sorts of EAs, where the list is in effect a population of promising visited solutions. By not storing the solutions themselves, LAHC stays a memory-affordable algorithm.

The idea of *late acceptance* can be implemented with minor variations and extensions. Building upon the work of Burke and Bykov [4], we settle on their final version whose pseudocode is reproduced as Algorithm 1.

An initial current solution is randomly generated and all the L_h elements of the history list are initialized with the same value: the cost of the current solution. The algorithm combines the late acceptance idea with the greedy rule of always accepting non-worsening moves as done is SA, i.e. it accepts a candidate solution if it is better than that solution which was current L_h iterations before or if it is not worse than the current solution of the immediate previous iteration. Another extension proposed by Burke an Bykov [4] is to update the history list with better values only and exclude any updating with worse values. The updates on the list are constant-time operations due to a circular list implementation (Line 14 of Algorithm 1), therefore not requiring actual removal and element shifting. Any stopping criterion can be used but here we follow exactly what was suggested in [4], i.e. the algorithm halts when the number of consecutive non-improving (idle) iterations reaches 2 percent over the total number of iterations, and at least 100 thousand iterations are performed to avoid early termination. Please refer to the pseudocode in Algorithm 1 for all the details.

2.2 Parameter-less search

Results reported by Burke and Bykov [4] suggest that the history length parameter has a more meaningful interpretation from the user point of view, as it is directly related to solution quality and execution time. Specifically, the longer the history length is, the slower the algorithm becomes in reaching a good solution quality. But also, the longer the history length is, the better chance the algorithm has of reaching a better quality solution given enough time do so. The tradeoff is a logical one; to reach a better solution quality we should expect to pay a price in terms of algorithm runtime.

This tradeoff is akin to what is observed with respect to population sizing in EAs. In general, an EA with a large population has a better chance, given sufficient time, to reach a better solution quality than the same EA with a smaller population [9]. On the other hand, all other things being equal, an EA with a large population is slower than the same EA with a smaller population. Effective methods for automating the population size parameter of an EA have been proposed in the literature. In particular, restarting an EA with an exponentially increasing population size has been suggested by Harik and Lobo [10] and by Smorodkina and Tauritz [16] in the context of genetic algorithms, and by Auger and Hansen [1] with respect to the Covariance Matrix Adaptation Evolution Strategy (CMA-ES). The same technique has been used successfully with model-based EAs such as the Hierarchical Bayesian Optimization Algorithm (hBOA) [15], and more recently with the Gene-pool Optimal Mixing EA (GOMEA) family of algorithms [2, 5, 14].

Given that the population size parameter of an EA appears to have a similar effect (and involve similar tradeoffs) to the history length parameter of LAHC, it is likely that the methods used to automate population sizing in EAs can also be used to automate the history length parameter of the LAHC algorithm. It is precisely this observation that led us to conduct the research presented herein.

3 LAHC WITH EXPONENTIALLY INCREASING HISTORY LENGTH

To eliminate the need to specify the history length parameter, we propose an automated restart strategy for the LAHC algorithm with an exponentially increasing history length. The strategy follows closely what was done in [1]. Initially, the history length can be a very small number, for example $L_h = 1$. The restart is fired whenever the LAHC with a fixed history length L_h stops as dictated by Line 7 of Algorithm 1. The overall stopping criterion for the restart strategy has to be specified in some other way. Criteria that come to mind could be for example a total maximum number of iterations, a total maximum execution time, a certain solution quality reached, or a certain number of restarts without an improvement in the current solution at the end of each LAHC execution, to name a few. With respect to the rate of increase for the history length we use a factor of 2, i.e. the history length doubles on each restart. We call the resulting algorithm *Parameter-less LAHC* (pLAHC).

Another form of growth schedule could be specified, for example linear instead of exponential. However, we settle on exponential with a factor of 2 because it seems to be the most logical thing to do, following previous research by others [1, 2, 5, 10, 14, 15].

3.1 Experiments with LAHC alone

We start by presenting a series of experiments of the LAHC algorithm with fixed history length to confirm the existence of the tradeoff observed by Burke and Bykov [4]. We do it for several instances of the TSP taken from the well-known TSPLIB repository available at http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. We use the exact same instances and the exact same TSP perturbation operator used in [4], namely the double-bridge move introduced by Lin and Kernighan [13] where a tour is randomly divided into two parts and subsequently reconnected in reverse order. For completeness, Table 1 shows the names and sizes (number of cities) of the TSP instances used.

Table 1: TSP benchmark instances taken from the TSPLIB repository.

Dataset	Size
rat783	783
u1060	1060
fl1400	1400
u1817	1817
d2103	2103
pcb3038	3038
fl3795	3795

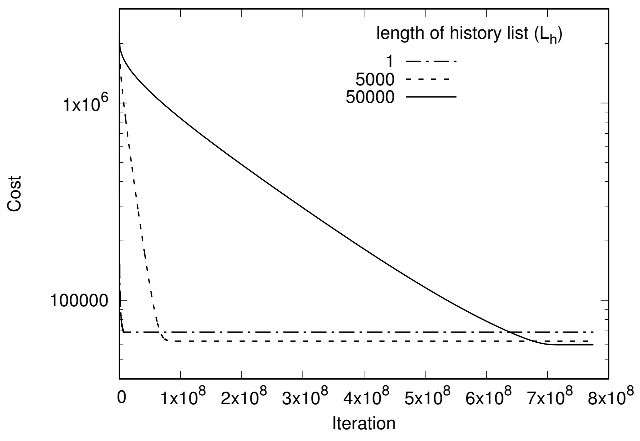


Figure 1: Tradeoff between solution quality and number of iterations, for the u1817 instance. Plot obtained with averaged data collected from 100 independent runs. Solution quality (cost) is shown in log scale.

For every instance, LAHC is run for 100 independent times and the results are averaged. We use 3 settings for the history length parameter (1, 5000, and 50000), replicating the experiments conducted in [4]. Table 2 shows the results obtained. The Cost column gives the best ever solution quality reached by the algorithm by the end of the run, averaged over the 100 runs. The Iterations column gives the number of iterations executed until the end of the run, averaged over the 100 runs.

The results confirm the observations made by Burke and Bykov. The longer the history length, the slower the algorithm is but a better solution quality is reached. Figure 1 shows the above mentioned tradeoff for the u1817 instance. For the other instances the pattern observed is similar.

3.2 Experiments with automated restarts

We now present experiments of the parameter-less implementation of LAHC as described at the beginning of Section 3, i.e. with automated restarts doubling the history length L_h on each restart. Initially $L_h = 1$. We run the algorithm on the same TSP instances

Table 2: Results for seven TSP instances produced by LAHC L_h of 1, 5000, and 50000. The results are averaged over 100 independent runs, and match very well with those reported by Burke and Bykov.

Dataset	$L_h = 1$		$L_h = 5000$		$L_h = 50000$	
	Cost	Iterations	Cost	Iterations	Cost	Iterations
rat783	10808	774187	9354	28375627	9105	258717906
u1060	264264	2177387	235426	43375332	229013	389063282
fl1400	22483	4055588	20732	57679069	20426	492849859
u1817	69056	8384640	62175	90558784	59502	750645922
d2103	98643	12444579	89579	112400626	86370	877424569
pcb3038	160238	27520354	150439	176549830	144211	1342738864
fl3795	33159	63452752	31147	264306010	30170	1816215196

described earlier. Again, 100 independent runs are executed. The stopping criterion for each run is reaching the target solution quality C_x obtained by the regular LAHC with history length x. The results are summarized in Table 3. As an example, the entry in Table 3 corresponding to the instance rat783 and C_1 has the value 1116839. This value is the number of iterations needed by pLAHC to reach the solution quality of 10808 (which can be read from Table 2, rat783 with $L_h = 1$) on all of the 100 independent runs.

As expected, pLAHC is slower than a tuned LAHC (see the overhead factors in Table 3), however, pLAHC needs no tuning. More importantly, pLAHC tries to escape local optima by restarting with a larger history length when its current history length appears to be insufficient. LAHC, on the other hand, needs to have L_h properly set; not doing so makes it unable to improve beyond a certain solution quality. For example, if LAHC is allowed to run for more iterations (ignoring the 2 percent idle iteration condition) it would not improve the solution quality that much more. Take for example the rat783 instance. If we run LAHC on it, with $L_h = 1$, for 710536424 iterations (those needed by pLAHC to reach a solution quality $C_{50000} = 9105$) the average solution quality over 100 independent runs is only 9933, nowhere close to the 9105 value. As a matter of fact, none of those 100 runs reaches the 9105 value. A similar behaviour occurs with the other instances.

Our experiments collect other relevant information concerning the pLAHC runs. Due to space restrictions we focus our analysis on a specific instance, u1817, which is median in terms of dimensionality (number of cities). Figure 2 shows the distribution of the required history length needed by pLAHC to reach the target solution quality (C_1, C_{5000}, C_{50000}) corresponding to the u1817 instance. In Figure 2a we can observe that the majority of the runs (56%) reach the target solution quality not needing any restarts, i.e. with $L_h = 1$. This result is consistent with what one would expect because we are only running pLAHC until it reaches the target quality C_1 (obtained by LAHC with $L_h = 1$). Similarly, in Figure 2b we observe that most of the pLAHC runs reach the target solution quality C_{5000} (obtained by LAHC with $L_h = 5000$) either with $L_h = 4096$ (27%) or $L_h = 8192$ (67%). Again, this is consistent with what is expected because these are the values closest to 5000. The same expected behaviour occurs in Figure 2c for the C_{50000} case, with most pLAHC runs needing to reach a history length of 65536, close to 50000.

Table 3: Number of iterations needed by pLAHC to reach at least the same solution quality as that obtained by LAHC. OF stands for overhead factor, i.e. how much slower pLAHC is compared with LAHC. The results are averaged over 100 independent runs.

Dataset	Iterations needed by pLAHC to reach quality C_x					
	C_1	OF	C_{5000}	OF	C_{50000}	OF
rat783	1116839	1.44	102151765	3.60	710536424	2.75
u1060	3352705	1.54	149197449	3.44	1108126907	2.85
fl1400	6267468	1.55	187044984	3.24	1042146093	2.11
u1817	13169262	1.57	352278209	3.89	2207268273	2.94
d2103	19253166	1.55	468100823	4.16	2572121231	2.93
pcb3038	42243384	1.53	804089579	4.55	4116303379	3.07
fl3795	104336892	1.64	1409994608	4.33	6302508196	3.47

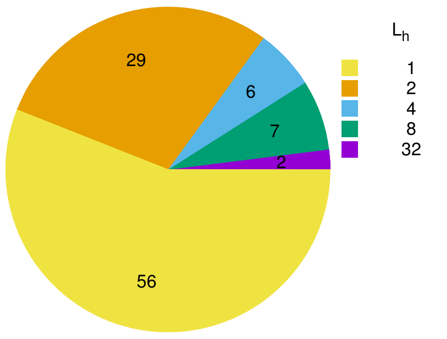
These results suggest that pLAHC is capable, without any tuning, to automatically discover an appropriate history length required to reach a certain target solution quality.

4 SPEEDING UP WITH SEEDED RESTARTS

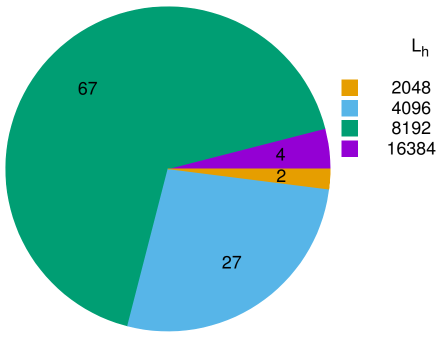
Here we explore a refinement of the pLAHC algorithm presented in the previous section. The idea is to use information collected from the execution of a LAHC run to seed the next history list upon a restart. In other words, as opposed to having the history list initialized with the solution quality of a randomly generated solution, we are going to explore an alternative mechanism to avoid starting the search from scratch. We note that the idea of seeding the start of the search in the context of parameter-less search algorithms has been suggested before (see [11]).

The first thing that comes to mind is to use the current solution of the LAHC run that has just expired, and use its cost function value to initialize the history list of the new LAHC run. We tested this idea but the results were not good. The reason is rather obvious. If the history list is filled up with the cost value of the current solution, there is very little chance that the delayed acceptance criteria is successful because the current solution is already very good (a local optimum) and the history list has no diversity; the combination of these two factors leads to having most perturbations rejected.

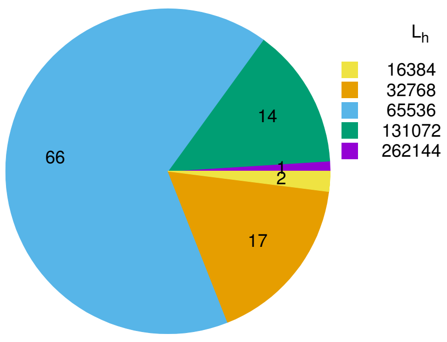
The idea behind *late accepting* is to use a delayed comparison (with a solution which was current several iterations ago) to escape



(a) C_1



(b) C_{5000}



(c) C_{50000}

Figure 2: Distribution of the required history length needed by pLAHC to reach the target solution quality (C_1, C_{5000}, C_{50000}) corresponding to the u1817 instance.

from a local optima more easily. It is therefore desirable that the history list cannot be composed of extremely good and identical values; it needs to have diversity. Note that no diversity with a low quality value would not be a problem because in that case most perturbations on a current solution would be accepted (but that corresponds to the standard initialization of LAHC, i.e. to start the search from scratch, which is what we are trying to avoid).

Algorithm 2: Parameter-less Late Acceptance Hill-Climbing with Seeding (pLAHC-s).

```
Output: A solution to a given problem.
Produce an initial solution $s$                 // Usually at random
Calculate its cost function value $C(s)$
$SIL = []$                                      // Successful iteration list
Append $C(s)$ to $SIL$
$L_h = 1$
$best = s$
while stopping criterion is not true do
    $v = 0$
    while $v < L_h$ do                          // History list initialization
        $w = SIL.size() - 1 - (v \textbf{ mod } SIL.size())$
        $f_v = SIL_w$
        $v = v + 1$
    sort $(f)$
    $I = 0$                                     // Iteration counter
    $I_{idle} = 0$                              // Idle iteration counter
    do until $(I > 100000)$ and $(I_{idle} > I \times .02)$
        Construct a candidate solution $s^*$
        Calculate its cost function value $C(s^*)$
        if $C(s^*) \geq C(s)$ then
            $I_{idle} = I_{idle} + 1$
        else
            $I_{idle} = 0$                      // Reset counter
        $v = I \textbf{ mod } L_h$              // Virtual beginning
        if $C(s^*) < f_v$ or $C(s^*) \leq C(s)$ then
            $s = s^*$                           // Accept candidate
        else
            $s = s$                             // Reject candidate
        if $C(s) < f_v$ then
            $f_v = C(s)$                        // Update the fitness array
        $I = I + 1$
        if $C(s) < SIL_{SIL.size()-1}$ then     // Update SIL
            Append $C(s)$ to $SIL$
            $best = s$
    $L_h = 2 \times L_h$
    $s = best$
return $s$
```

The above argument suggests that to be successful, a seeding strategy needs to initialize the history list with good, yet diverse, cost function values of previously visited solutions. The very next thing that comes to mind is to fill the history list with the cost function values of past successful iterations. By past successful iterations, we mean those iterations that yielded solutions that improve upon the best cost function value ever found so far. To do so we use another list to memorize those best-so-far values. We call this list the *successful iterations list* (SIL).

pLAHC is now augmented with a *successful iterations list*. The list is shared among the LAHC restarts, and gets updated whenever there's an improvement on the best-so-far solution quality value, regardless of which LAHC produces it. Once a current LAHC expires, the next LAHC restarts with its history list initialized with values taken from the end of *successful iterations list*. In case the length of the *successful iterations list* is less than the length of the newly created LAHC history list, we keep looping through the *successful iterations list* until the history list is filled up (in that case, we will have duplicate values). Lines 8-12 of Algorithm 2 denote this initialization. The history list is then sorted with larger values at the virtual beginning and smaller values at the virtual end of the list.

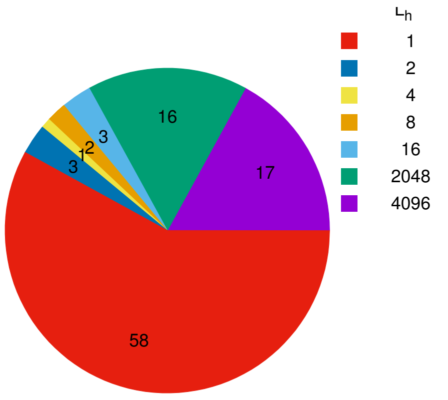
This way, instead of restarting the search from a randomly generated solution, LAHC restarts from a good solution found previously and hopefully has sufficient diversity of good solution quality values in its history list right from the beginning. As we shall see, the combination of these two factors gives enough flexibility for LAHC to take advantage of the *late acceptance* idea without needing to start anew on every restart. We refer to this refinement of pLAHC as *Parameter-less LAHC with seeding* (pLAHC-s). Pseudocode for it is presented in Algorithm 2, with lines in red color indicating changes with respect to Algorithm 1 (LAHC) to design pLAHC-s.

4.1 Experiments with pLAHC-s

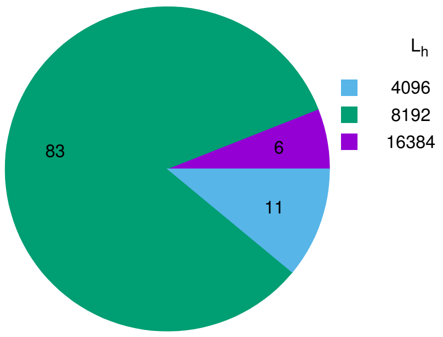
We repeat the experiments described in Section 3.2, this time with pLAHC-s instead of pLAHC. The results are summarized in Table 4. It is straightforward to observe that seeding the history list speeds up the search considerably. The overhead factors of pLAHC-s with respect to LAHC are substantially lower than those obtained with pLAHC, especially for the target solution quality C_{5000} and C_{50000}. These are the cases where LAHC requires long history length values, i.e. where restarts are an absolute must.

Figure 3 is the equivalent of Figure 2 for the pLAHC-s case. It shows the distribution of the required history length needed by pLAHC-s to reach the target solution quality (C_1, C_{5000}, C_{50000}) corresponding to the u1817 instance. Again, the results suggest that pLAHC is capable, without any tuning, to automatically discover an appropriate history length required to reach a certain target solution quality.

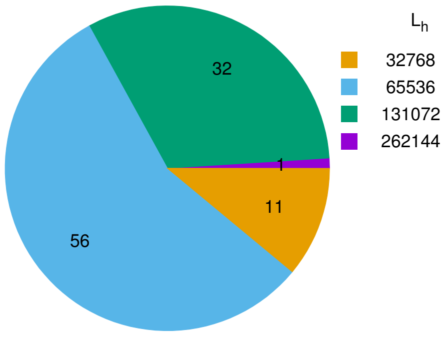
Figure 4 illustrates the dynamics of restarts as the search progresses through time, pLAHC and pLAHC vis-a-vis, for the u1817 instance. Figure 4a is for the target solution quality C_1, Figure 4b for target quality C_{5000}, and Figure 4c for target quality C_{50000}. The data used in the various plots was collected from the runs described earlier. For any given point in time (iteration number on the horizontal axis) we obtain the history list length (averaged over 100 independent runs) at that given point in time, for both pLAHC and pLAHC-s. It is notable that pLAHC-s quickly eliminates small-sized history lengths compared to pLAHC. The results agree with our



(a) C_1

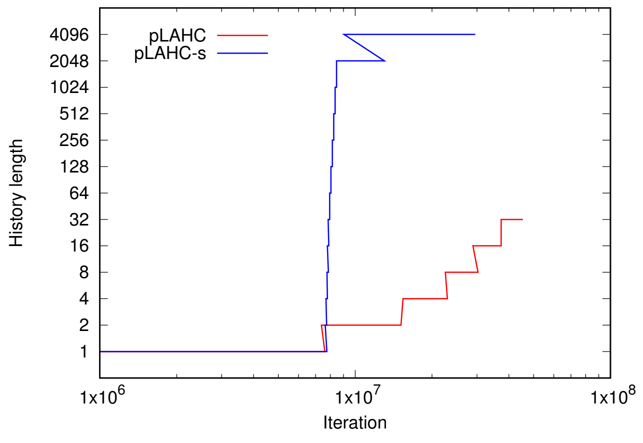


(b) C_{5000}

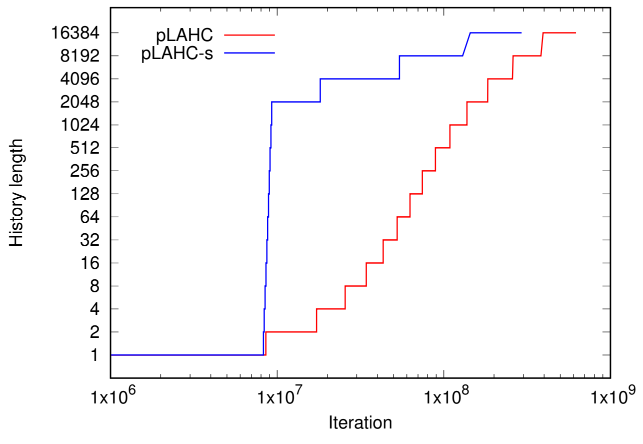


(c) C_{50000}

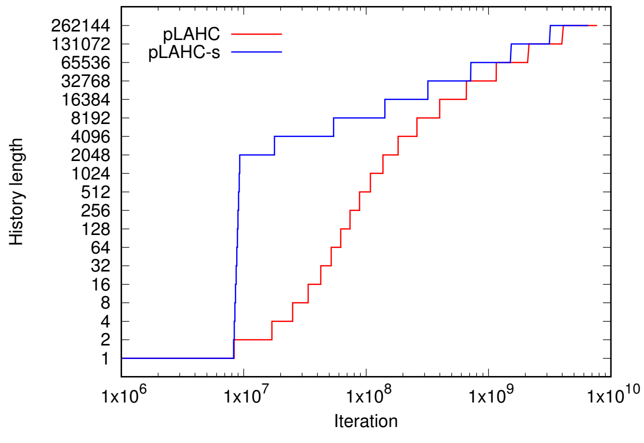
Figure 3: Distribution of the required history length needed by pLAHC-s to reach the target solution quality (C_1, C_{5000}, C_{50000}) corresponding to the u1817 instance.



(a) C_1

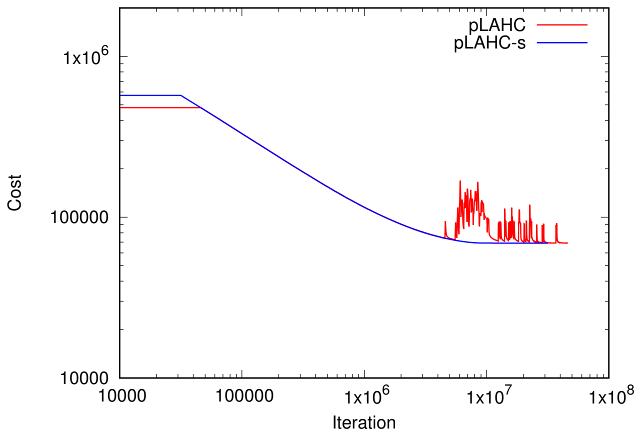


(b) C_{5000}

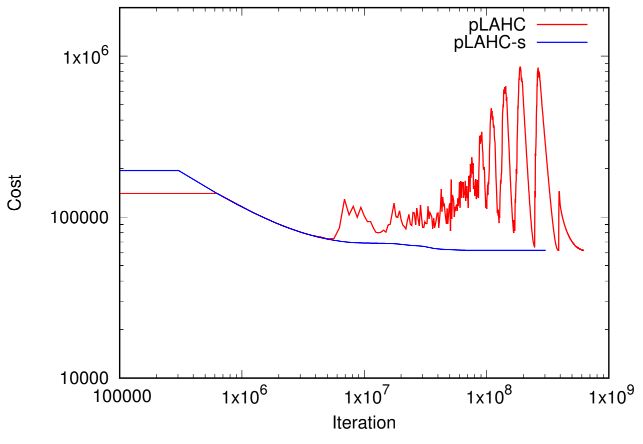


(c) C_{50000}

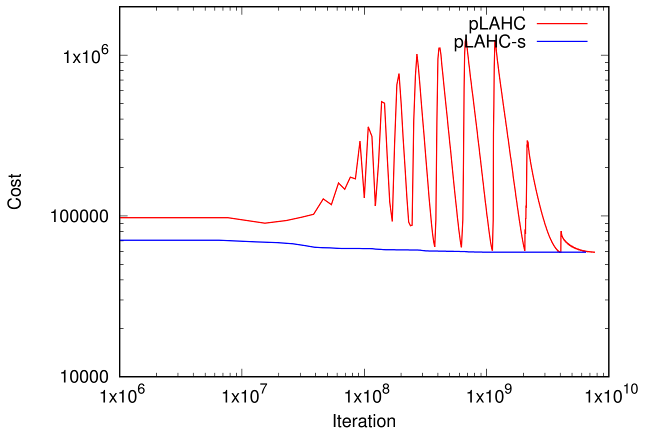
Figure 4: Average history list length through time for the u1817 instance, pLAHC and pLAHC-s vis-a-vis. Data collected from the average of 100 independent runs.



(a) C_1



(b) C_{5000}



(c) C_{50000}

Figure 5: Average current solution cost through time for the u1817 instance, pLAHC and pLAHC-s vis-a-vis. Data collected from the average of 100 independent runs.

Table 4: Number of iterations needed by pLAHC-s to reach at least the same solution quality as that obtained by LAHC. OF stands for overhead factor, i.e. how much slower pLAHC-s is compared with LAHC. The results are averaged over 100 independent runs.

Dataset	Iterations needed by pLAHC-s to reach quality C_x					
	C_1	OF	C_{5000}	OF	C_{50000}	OF
rat783	771194	1.00	56110157	1.98	670283547	2.59
u1060	2225981	1.02	80241114	1.85	993939929	2.55
fl1400	6620509	1.63	136349568	2.36	987121266	2.00
u1817	13083187	1.56	130230000	1.44	1977770681	2.63
d2103	19857202	1.60	175890001	1.56	1718555845	1.96
pcb3038	48826067	1.77	163564379	0.93	2851442286	2.12
fl3795	128646608	2.03	360440032	1.36	3844065351	2.12

intuition; it is the quick elimination of small-sized history lengths that allows pLAHC-s to achieve a lower overhead factor compared to pLAHC.

Figure 5 also shows interesting dynamics of pLAHC and pLAHC-s on the u1817 instance, this time showing the cost of the current solution through time, averaged over 100 independent runs. Note how the cost oscillates in the case of pLAHC but not in the case of pLAHC-s. The oscillation observed with the pLAHC case is due to restarting every single LAHC from scratch. This doesn't occur with pLAHC-s, precisely because of the seeding mechanism. Overall, the plots shown in Figure 4 and Figure 5 help us to understand in a clear way why the seeding mechanism is beneficial within the parameter-less restart strategy.

Although the plots in Figures 2, 3, 4, and 5, only concern the u1817 instance, we did analyze what happens with the other instances and the pattern observed is similar to the u1817 case.

The source code in C++ for the implementation of LAHC, pLAHC, and pLAHC-s, described in this paper, is submitted as supplementary material, along with the input files necessary to replicate all experiments presented herein. It is also available online at https://github.com/mbazargani/pLAHC.

5 SUMMARY AND CONCLUSIONS

This paper proposed an automated strategy to eliminate the sole parameter of the LAHC algorithm, a general purpose metaheuristic introduced by Burke and Bykov [4]. LAHC belongs to the class of local search methods, is simple to implement, and easy to use in practice, requiring the tuning of a single parameter whose effect on the search is well understood. Building on techniques that have been successfully used in parameter-less evolutionary algorithms, we developed a parameter-less version of LAHC, and then refined it even further with seeded restarts. The resulting parameter-less algorithm is of course slower than a LAHC fine-tuned with an appropriate history length value, but has the advantage of not requiring any tuning (which in practice is time consuming, and needs to be redone for different problems, and even for different instances across the same problem). Experiments on several TSP benchmark instances show that the parameter-less LAHC is effective.

It is noteworthy to observe that the parameter-less techniques developed earlier in the context of EAs are also applicable in the context of local search methods, showing that those techniques have a broad applicability.

ACKNOWLEDGMENTS

The authors would like to thank John H. Drake for providing useful comments on this paper.

REFERENCES

[1] Anne Auger and Nikolaus Hansen. 2005. A restart CMA evolution strategy with increasing population size. In *The 2005 IEEE International Congress on Evolutionary Computation (CEC'05)*, B. McKay and others (Eds.), Vol. 2. 1769–1776.

[2] Peter A. N. Bosman, Ngoc Hoang Luong, and Dirk Thierens. 2016. Expanding from Discrete Cartesian to Permutation Gene-pool Optimal Mixing Evolutionary Algorithms. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, Denver, CO, USA, July 20 - 24, 2016*, Tobias Friedrich, Frank Neumann, and Andrew M. Sutton (Eds.). ACM, 637–644. DOI: http://dx.doi.org/10.1145/2908812.2908917

[3] Edmund K. Burke and Yuri Bykov. 2008. A late acceptance strategy in hill-climbing for exam timetabling problems. In *Proceedings of the 7th International Conference on the Practice and Theory of Automating Timetabling (PATAT 2008)*. extended abstract.

[4] Edmund K. Burke and Yuri Bykov. 2017. The late acceptance Hill-Climbing heuristic. *European Journal of Operational Research* 258, 1 (2017), 70–78. DOI: http://dx.doi.org/10.1016/j.ejor.2016.07.012

[5] Willem den Besten, Dirk Thierens, and Peter A. N. Bosman. 2016. The Multiple Insertion Pyramid: A Fast Parameter-Less Population Scheme. In *Parallel Problem Solving from Nature - PPSN XIV - 14th International Conference, Edinburgh, UK, September 17-21, 2016, Proceedings (Lecture Notes in Computer Science)*, Julia Handl, Emma Hart, Peter R. Lewis, Manuel López-Ibáñez, Gabriela Ochoa, and Ben Paechter (Eds.), Vol. 9921. Springer, 48–58. DOI: http://dx.doi.org/10.1007/978-3-319-45823-6_5

[6] Gunter Dueck. 1993. New Optimization Heuristics: The Great Deluge Algorithm and the Record-to-Record Travel. *J. Comput. Phys.* 104, 1 (1993), 86 – 92. DOI: http://dx.doi.org/10.1006/jcph.1993.1010

[7] Gunter Dueck and Tobias Scheuer. 1990. Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing. *J. Comput. Phys.* 90, 1 (1990), 161 – 175. DOI: http://dx.doi.org/10.1016/0021-9991(90)90201-B

[8] Fred Glover. 1989. Tabu Search: Part I. In *ORSA Journal of Computing*, Vol. 1. 190–206.

[9] Georges R. Harik, Erick Cantú-Paz, David E. Goldberg, and Brad L. Miller. 1999. The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation* 7, 3 (1999), 231–253.

[10] Georges R. Harik and Fernando G. Lobo. 1999. A parameter-less genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*, W. Banzhaf and others (Eds.). Morgan Kaufmann, San Francisco, CA, 258–265.

[11] Ekaterina Holdener. 2008. *The art of parameterless evolutionary algorithms.* Ph.D. Dissertation. Missouri University of Science and Technology, Rolla, MO, USA.

[12] Scott Kirkpatrick, C. D. Gelatt, and Mario P. Vecchi. 1983. Optimization by Simulated Annealing. *Science, Number 4598, 13 May 1983* 220, 4598 (1983), 671–680.

[13] S. Lin and Brian W. Kernighan. 1973. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Oper. Res.* 21, 2 (April 1973), 498–516. DOI: http://dx.doi.org/10.1287/opre.21.2.498

[14] Hoang Ngoc Luong, Han La Poutré, and Peter A. N. Bosman. 2015. Exploiting Linkage Information and Problem-Specific Knowledge in Evolutionary Distribution Network Expansion Planning. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015*, Sara Silva and Anna Isabel Esparcia-Alcázar (Eds.). ACM, 1231–1238. DOI: http://dx.doi.org/10.1145/2739480.2754682

[15] Martin Pelikan and Tz-Kai Lin. 2004. Parameter-Less Hierarchical BOA. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004), Part II, LNCS 3103*, K. Deb and others (Eds.). Springer, 24–35.

[16] Ekaterina Smorodkina and Daniel R. Tauritz. 2007. Greedy Population Sizing for Evolutionary Algorithms. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2007.* IEEE, 2181–2187.