EAI Endorsed Transactions
on Industrial Networks and Intelligent Systems
Research Article EAI.EU

Optimizing the Modified Lam Annealing Schedule

Vincent A. Cicirello*

Computer Science, Stockton University, 101 Vera King Farris Drive, Galloway, NJ 08205

Abstract

Simulated annealing is a metaheuristic commonly used for combinatorial optimization in many industrial applications. Its runtime behavior is controlled by an algorithmic component known as the annealing schedule. The classic annealing schedules have control parameters that must be set or tuned ahead of time. Adaptive annealing schedules, such as the Modified Lam, are parameter-free and self-adapt during runtime. However, they are also more complex than the classic alternatives, leading to more time per iteration. In this paper, we present an optimized variant of Modified Lam annealing, and experimentally demonstrate the potential significant impact on runtime performance of carefully optimizing the annealing schedule.

Received on 07 October 2020; accepted on 03 December 2020; published on 16 December 2020

Keywords: simulated annealing, modified Lam, self-adaptive, parameter-free, combinatorial optimization

doi:10.4108/eai.16-12-2020.167653

1. Introduction

Simulated Annealing (SA), introduced by Kirkpatrick *et al.* (1983), is a metaheuristic commonly used for combinatorial optimization in many industrial applications (Delahaye *et al.*, 2019). The behavior of SA is inspired by the annealing process commonly used in metallurgy as well as in the manufacture of glass. The physical annealing process involves first heating the metal or glass to a high temperature, forming it as desired, and then allowing it to cool slowly. A slow rate of cooling is important in order to reduce internal stress and to increase stability of the final result.

SA begins with a random solution to a combinatorial optimization problem, and is initialized with a high value of a control parameter referred to as "temperature." This temperature is then adjusted during the run by a component of the SA called the annealing schedule. Each iteration of SA involves generating a random neighbor of the current solution. A random decision is then made to determine whether to accept that neighbor or to reject it. If it is accepted, the neighbor replaces the current solution. The acceptance decision depends upon the quality of the neighbor, as well as the temperature parameter. At very high values of temperature, neighbors are likely to be accepted even if their value on the optimization objective is not good. This is similar to the physical annealing process where at high temperatures one can change the shape of the metal or glass easily. Later in the run of SA, when the temperature is lower, SA is less likely to accept a neighbor if the current solution is better, settling in upon a locally optimal solution. This is similar to the physical process where internal stresses are minimized in the final product resulting in a stable form.

An "annealing schedule" controls how SA's temperature parameter changes during the search. In the classic annealing schedules, the temperature is monotonically decreasing, and those schedules in turn are controlled by one or more parameters, which must be set or tuned ahead of time. This is the most critical part of an SA. If the temperature cools too slowly, the search spends too much time in a random walk, requiring much longer to settle in upon a local optima. If the temperature cools too quickly, the search may produce a less desirable local optima. There are a variety of adaptive annealing schedules that have been proposed in the literature to deal with this issue (Boyan, 1998; Hubin, 2019; Štefankovič *et al.*, 2009). Rather than strictly monotonically decreasing, the adaptive annealing schedules adjust temperature both up and down based on feedback from the search. We discuss both classical and adaptive annealing schedules in Section 2, as well as other related work.

*Corresponding author. Email: vincent.cicirello@stockton.edu. Website: https://www.cicirello.org/

In this paper, in Section 3, we propose an optimized variant of Modified Lam, an adaptive annealing schedule. We have integrated this optimized Modified Lam into the open-source Java library, Chips-n-Salsa (Cicirello, 2020), which is a library of stochastic local search algorithms. The source code of the Chips-n-Salsa library is maintained on GitHub (https://github.com/cicirello/Chips-n-Salsa), and releases can be downloaded and installed from Maven Central (https://search.maven.org/artifact/org.cicirello/chips-n-salsa). The Chips-n-Salsa website (https://chips-n-salsa.cicirello.org/) contains API documentation and other details.

We conduct experiments on the impact that these optimizations have on SA runtime. The results of those experiments are discussed in Section 4. In the first part of the experiments, we isolate the annealing schedule and compare the runtime of the original and optimized versions of the Modified Lam annealing schedule independent of SA. Then, in the second part of the experiments, we investigate the impact on SA runtime for several optimization problems with a variety of solution representation. Additionally, to enable reproducibility, we provide the source code of our experiments, the source code used in analyzing the experimental data, as well as the raw data itself, in a GitHub repository (https://github.com/cicirello/modified-lam-experiments). We offer conclusions in Section 5.

2. Background and Related Work

2.1. Basics of Simulated Annealing

Algorithm 1 shows pseudocode for SA. It is expressed assuming that we are minimizing the cost function $C(S)$. SA begins by generating a random initial solution S to the problem (line 6). It also must initialize the temperature T to some initial temperature T_0 (line7), which should be high at the start of the search. Just how high depends upon the scale of the cost function that we are minimizing. It is usually set by the implementer to a constant, but one may also randomly sample the solution space as a basis for choosing T_0.

Each iteration of the main loop (lines 8–14) begins by generating a random neighbor S' of the current solution S (line 9). In the pseudocode $\eta(S)$ is the neighborhood function. The neighborhood function generates random neighbors, where a neighbor is a similar solution to the current one. For example, if the solution S is represented by a permutation (e.g., for an ordering problem like the Traveling Salesperson), then a neighbor might have a pair of random elements swapped. SA must then decide whether or not to "accept" the neighbor S'. This decision depends upon the relative cost of S' and S, as well as the temperature T. If the cost $C(S')$ is lower than the cost $C(S)$, then

Algorithm 1 Simulated Annealing

```
1:  ## Notation:
2:  ## N is the run length in number of evaluations.
3:  ## C(S) is the real-valued cost of solution S.
4:  ## η(S) is the set of neighbors of solution S.
5:  ## U() generates a uniform random value in [0, 1).
6:  S ← GenerateRandomInitialState
7:  T ← T_0
8:  for i = 1 to N do
9:      S' ← random selection from η(S)
10:     if C(S') ≤ C(S) or U() < e^((C(S)−C(S'))/T) then
11:         S ← S'
12:     end if
13:     T ← f(T)
14: end for
15: return Best solution found during run
```

the neighbor is definitely accepted. Otherwise, if the neighbor S' is more costly, a stochastic decision is made, and with probability $e^{(C(S)-C(S'))/T}$ it is accepted, which is known as the Boltzmann distribution. This acceptance probability decreases as T decreases, and also decreases as $C(S')$ increases. If the neighbor is accepted, SA replaces S with S', and otherwise discards S'.

At the very end of each iteration T is updated (line 13) with $T \leftarrow f(T)$. The most common form of $f(T)$ is "exponential cooling":

$$f(T) = \alpha \cdot T, \tag{1}$$

where $\alpha \in [0, 1]$ is a constant cooling rate that is typically set close to 1 (e.g., 0.9, 0.95, 0.99, etc). In this way, if the search runs long enough, T will eventually become 0. The next most common annealing schedule is "linear cooling":

$$f(T) = T - \lambda, \tag{2}$$

where λ is a constant. Care must be taken to choose λ relative to T_0 and the run length N such that T never becomes negative.

SA finally returns the best solution encountered during the run (line 15).

2.2. Applications of Simulated Annealing

SA has been used for a very large variety of problems. Although most commonly used in discrete optimization problems, it can also be used for real-valued function optimization (e.g., using Gaussian mutation (Hinterding, 1995) as the neighborhood function). It is most commonly used for NP-Hard combinatorial optimization problems.

There are many industrial applications of SA. For example, Dinh *et al.* (2019) use SA for an assembly

line balancing problem. Zhichao *et al.* (2018) use SA to detect foreign fibers in images of cotton to enable removal of those foreign fibers. Liang *et al.* (2018) optimize the placement of bus stops in an urban bus transportation system using SA. Hu *et al.* (2019) use SA for an FPGA placement problem. Ma *et al.* (2019) apply SA to path planning in optical fiber transmission networks. Cismaru (2018) optimizes energy efficiency of a train for a predetermined route using SA.

SA has been applied to a variety of problems related to wireless sensor networks. Daryanavard and Harifi (2019) use SA for UAV path planning, where the UAVs must navigate to collect information from a network of wireless sensors. Sun and Zhang (2018) use SA for a wireless sensor network placement problem. Li *et al.* (2019) use a combination of SA and particle swarm optimization for underwater acoustic positioning.

SA has even been used in software testing. For example, Yan *et al.* (2019) consider a software product line testing problem, and use a hybrid of SA and a genetic algorithm to generate a test suite while optimizing test coverage. Zamli *et al.* (2018) investigate the large increase in size of test cases that can happen over time in a software project, and use SA to search for and eliminate redundancy among test cases.

2.3. Advanced Simulated Annealing Concepts

Adaptive Annealing Schedules: Earlier in Section 2.1, we saw the basic form of SA in Algorithm 1, and in particular we considered the two most common annealing schedules, exponential cooling and linear cooling. The challenge with using one of these annealing schedules is in effectively choosing the parameter values. If you cool the temperature too quickly, the search will reach a local optima too soon and stagnate. If you cool the temperature too slowly, SA will spend too much time in a random walk.

An alternative to trying to tune the control parameters a priori is to instead use an annealing schedule that is self-adaptive. The earliest example of this is the work of Lam and Delosme (1988) where they studied the rate that neighbors are accepted during optimal runs of SA. In particular, they observed that during the first 15% of an optimal run of SA that the rate of neighbor acceptance decreases exponentially from 100% at the start of the run to 44%. The acceptance rate then remains approximately constant for 50% of the run, and then finally decays exponentially during the last 35% of the run. Following their observations, Lam and Delosme (1988) devised an approach to exploit their observations where they track the acceptance rate during the run, and modify the neighborhood function in an attempt to match the theoretical rate of acceptance. For example, if SA is accepting too many neighbors, they decrease the size of the local neighborhood; and if SA is accepting too few neighbors, they increase the size of the local neighborhood.

Modifying the neighborhood function in SA is not always practical. Swartz (1993) developed, and Boyan (1998) refined, what is now known as the Modified Lam annealing schedule, named after the work of Lam and Delosme (1988). In the Modified Lam, the neighborhood function is not changed during the search. Instead, the temperature T is adjusted up or down in order to track the Lam acceptance rate, whereas the original work of Lam and Delosme (1988) used a monotonically decreasing T. It is actually much more straightforward to adjust T to influence the acceptance rate than it is to adjust neighborhood size. Increasing T increases acceptance rate, and decreasing T decreases acceptance rate. Complete details of the Modified Lam annealing schedule, including pseudocode, can be found in Section 3.

The Modified Lam is not the only adaptive annealing schedule. There are other examples in the literature (e.g. Hubin, 2019; Štefankovič *et al.*, 2009). But we leave the reader to consult the references for other examples, since our focus in this paper is specifically on the Modified Lam.

Restart Schedules: In other forms of local search, such as hill climbers, it is common to use a restart approach where you run the local hill climber several times and return the best solution found across the restarts. Although restarting is sometimes used with SA, it is not as common. The reason is that many have shown that it is more effective to use a single long run of SA than it is to use an approach involving a best of several restarts, provided the annealing schedule is tuned appropriately for the longer run. However, it is not always feasible to determine how much time is available for the run, so you will encounter applications of SA using simple fixed length restarts.

In our own prior work (Cicirello, 2017), we investigated how we might deal with the uncertainty of available time for problem solving, and developed an approach that uses restarts to adapt the run length. Rather than choosing a run length ahead of time, we developed a schedule of run lengths called Variable Annealing Length (VAL) that begins with a very short run of SA. With VAL, each restart is a run of a length that is double that of the previous run. This effectively balances the risk associated with tuning the annealing schedule for a longer run than you actually have time for. The early very short runs find adequate solutions, and the quickly increasing run lengths lead to progressively better solutions as the SA is restarted. In that work, we showed that VAL's restart schedule of run lengths is significantly more effective than using fixed length restarts.

Parallel SA: There are a variety of approaches to parallel SA. Some parallel SA compute neighbor evaluations in parallel (e.g. Ludwin and Betz, 2011; Rudolph, 1993), such as the speculative moves of Ludwin and Betz (2011) in FPGA placement that optimizes the critical path length. Others use parallel computing to implement parallel multistart SA, such as approaches where the parallel instances of SA periodically share solutions (e.g. Jha and Menon, 2014; Ram *et al.*, 1996). In our own prior work on variable length restarts (Cicirello, 2017), we also introduced a parallel restart schedule called Parallel Variable Annealing Length (PVAL) that spreads the VAL restart schedule across multiple parallel instances of SA. Another approach to parallel SA is to decompose a problem into subproblems, and to then optimize the subproblems in parallel, such as the approach of Rahimian *et al.* (2015) for partitioning large social network graphs.

3. Optimizing Modified Lam Annealing

Original Modified Lam: Pseudocode for the original Modified Lam annealing schedule is shown in Algorithm 2, and is based on the description of the Modified Lam given by Boyan (1998).

The temperature T is initialized to 0.5 (see line 7). It mostly doesn't matter what T is initialized to in the Modified Lam, as it will be adjusted during the first several iterations until the acceptance rate matches the target acceptance rate, and then continue to be adjusted to track that target acceptance rate. The adjustment to T based on actual acceptance rate and the target acceptance rate is shown in lines 24–28. The actual acceptance rate is approximated using a reinforcement learning (RL) inspired technique (lines 13 and 15). For example, you can consider the 0.998 to be like a RL discount factor, and a reward of 0.002 when a neighbor is accepted (and 0 if it is not accepted). The target acceptance rate, referred to in the pseudocode as LamRate, follows the exponential decay for the first 15% and last 35% of the run in lines 18 and 20, respectively.

Optimized Modified Lam: The original Modified Lam annealing schedule computes a large number of exponentiations. Specifically, consider lines 18 and 20 of Algorithm 2. Those lines are inside the main loop. The exponentiation in line 18 is executed once for each of the iterations in the first 15% of the run; and the exponentiation in line 20 is executed once for each iteration in the final 35% of the run of SA. If the SA run is N iterations in length, then SA with the original Modified Lam annealing schedule computes $0.5N$ exponentiations. Exponentiation is a relatively expensive operation compared with the rest of the

Algorithm 2 Original Modified Lam Annealing

```
1:  ## Notation:
2:  ## N is the run length in number of evaluations.
3:  ## C(S) is the real-valued cost of solution S.
4:  ## η(S) is the set of neighbors of solution S.
5:  ## U() generates a uniform random value in [0, 1).
6:  S ← GenerateRandomInitialState
7:  T ← 0.5
8:  AcceptRate ← 0.5
9:  for i = 1 to N do
10:   S' ← random selection from η(S)
11:   if C(S') ≤ C(S) or U() < e^((C(S)−C(S'))/T) then
12:     S ← S'
13:     AcceptRate ← 0.998 · AcceptRate + 0.002
14:   else
15:     AcceptRate ← 0.998 · AcceptRate
16:   end if
17:   if i ≤ 0.15N then
18:     LamRate ← 0.44 + 0.56 · 560^(−i/(0.15N))
19:   else if i > 0.65N then
20:     LamRate ← 0.44 · 440^(−(i/N−0.65)/0.35)
21:   else
22:     LamRate ← 0.44
23:   end if
24:   if AcceptRate > LamRate then
25:     T ← 0.999T
26:   else
27:     T ← T/0.999
28:   end if
29: end for
30: return Best solution found during run
```

Modified Lam. Our aim with the Optimized Modified Lam is to minimize the number of exponentiations computed.

Our optimized version of the Modified Lam annealing schedule is shown in pseudocode form in Algorithm 3. We compute two exponentiations total (see lines 10 and 11) prior to entering the main loop. The exponentiations that had been inside the loop are each replaced with a multiplication (lines 21 and 24). Therefore, the $0.5N$ exponentiations of the original Modified Lam are replaced with two exponentiations and $0.5N$ multiplications.

Additionally, we can even cache the results of those two exponentiations, achieving an even larger time advantage over the original Modified Lam if we use restarts. The m_1 and m_2 (lines 10 and 11) only need to be recomputed during a restart if the run length N has changed. If all restarted runs are equal length, we can store these for reuse. For example, if we use a multistart approach where we run the SA some number of times R and return the best of the R solutions, then the original Modified Lam needs $0.5RN$ exponentiations,

Algorithm 3 Optimized Modified Lam Annealing

```
1:  ## Notation:
2:  ## N is the run length in number of evaluations.
3:  ## C(S) is the real-valued cost of solution S.
4:  ## η(S) is the set of neighbors of solution S.
5:  ## U() generates a uniform random value in [0, 1).
6:  S ← GenerateRandomInitialState
7:  T ← 0.5
8:  AcceptRate ← 0.5
9:  m_0 = 0.56
10: m_1 = 560^{-1/(0.15N)}
11: m_2 = 440^{-1/(0.35N)}
12: for i = 1 to N do
13:    S' ← random selection from η(S)
14:    if C(S') ≤ C(S) or U() < e^{(C(S)−C(S'))/T} then
15:       S ← S'
16:       AcceptRate ← 0.998 · AcceptRate + 0.002
17:    else
18:       AcceptRate ← 0.998 · AcceptRate
19:    end if
20:    if i ≤ 0.15N then
21:       m_0 = m_0 · m_1
22:       LamRate ← 0.44 + m_0
23:    else if i > 0.65N then
24:       LamRate ← LamRate · m_2
25:    else
26:       LamRate ← 0.44
27:    end if
28:    if AcceptRate > LamRate then
29:       T ← 0.999T
30:    else
31:       T ← T/0.999
32:    end if
33: end for
34: return Best solution found during run
```

whereas our optimized version still only requires 2 exponentiations (total across all R restarts) and $0.5RN$ multiplications.

Let us now confirm that the optimized version results in an equivalent sequence of target acceptance rates as that of the original version. In the pseudocode, the target acceptance rate is referred to as LamRate. Define LamRate(i) as the value of LamRate after update i, and consider the value of LamRate(i) during the first 15% of the run (i.e., when $i \leq 0.15N$). Begin by defining $m_0(i)$ as the value of m_0 after update i. The m_0 is initialized to 0.56 (see line 9), so $m_0(0) = 0.56$. During each update, m_0 is multiplied by m_1 (line 21). Therefore, $m_0(i)$ (for $i \leq 0.15N$) is as follows:

$$\begin{aligned} m_0(i) &= m_0(0) \cdot m_1^i \\ &= 0.56 \cdot (560^{-1/(0.15N)})^i \\ &= 0.56 \cdot 560^{-i/(0.15N)}. \end{aligned} \tag{3}$$

We can thus compute LamRate(i) in terms of $m_0(i)$ based on line 22 of the algorithm as follows:

$$\begin{aligned} \text{LamRate}(i) &= 0.44 + m_0(i) \\ &= 0.44 + 0.56 \cdot 560^{-i/(0.15N)}, \end{aligned} \tag{4}$$

which is precisely how LamRate is defined for the first 15% of the run in the original Modified Lam annealing schedule.

For the next 50% of the run, the target acceptance rate remains constant at 0.44 just like in the original version.

Next consider the last 35% of the run. The value of LamRate is adjusted (line 24) with a multiplication by m_2 (initialized in line 11) in this case. At the time of the first such adjustment in this phase of the algorithm, LamRate = 0.44. As before, define LamRate(i) as the value of LamRate after update i, but this time consider the last 35% of the run. We are specifically concerning ourselves with updates i, such that $0.65N < i <= N$. We can compute the value of LamRate(i) as follows:

$$\begin{aligned} \text{LamRate}(i) &= 0.44 \cdot m_2^{i-0.65N} \\ & 0.44 \cdot (440^{-1/(0.35N)})^{i-0.65N} \\ & 0.44 \cdot 440^{-i/(0.35N)+0.65/0.35} \\ & 0.44 \cdot 440^{(-i/N+0.65)/0.35} \\ & 0.44 \cdot 440^{-(i/N-0.65)/0.35}, \end{aligned} \tag{5}$$

which is precisely how LamRate is defined for the last 35% of the run in the original Modified Lam annealing schedule.

4. Experiments

4.1. Open Source Implementation

We implemented both the original Modified Lam and our optimized version within an open-source Java library, Chips-n-Salsa (release 2.2.0), of stochastic local search algorithms (Cicirello, 2020). The source code of Chips-n-Salsa is maintained on GitHub[1], and prebuilt releases are regularly deployed to Maven Central[2] as well as GitHub Packages[3] where from either of which the library can easily be imported into projects using common build tools such as Maven or Gradle (among others).

The Java class ModifiedLam is our implementation of our optimized version of the Modified Lam annealing schedule, and the class ModifiedLamOriginal is the original version. Both of these classes are found in the package org.cicirello.search.sa, which is where

[1] https://github.com/cicirello/Chips-n-Salsa
[2] https://search.maven.org/artifact/org.cicirello/chips-n-salsa
[3] https://github.com/cicirello?tab=packages

you will also find the class SimulatedAnnealing. The SimulatedAnnealing class is implemented to enable using the annealing schedule and neighborhood operator (referred to as mutation in the library) of your choice, and supports optimization of any type of structure. See the API documentation for complete details of functionality of Chips-n-Salsa[4].

The source code that implements all of our experiments for this paper is also available on GitHub (https://github.com/cicirello/modified-lam-experiments). This repository includes the source code of the Java programs implementing the experiments themselves, as well as a Python program used for the statistical analysis. The raw experimental data is found there as well. If you would like to reproduce the experiments on your own machine, see this GitHub repository for details.

We used Chips-n-Salsa 2.2.0 in the experiments, compiled on Ubuntu using OpenJDK 11, 64-bit, for a Java 8 target (the library currently supports Java 8 and up). Although we are the maintainers of this library, we used the released version rather than a development version to ensure reproducibility of our experimental results. The Java programs implemented to run the experiments were compiled on Windows 10 using OpenJDK 11 for a Java 11 target. The experiments were then run using the OpenJDK 64-bit Server VM (build 11.0.8+10) on a Windows 10 machine, with a AMD A10-5700 3.4 GHz CPU, and 8GB RAM.

4.2. Original vs Optimized Modified Lam

In this first experiment, we compare the original and optimized versions of the Modified Lam annealing schedule independent from SA. That is, we don't actually generate random neighbors. Instead, we generate a deterministic sequence of fake cost values. In this way, nearly all of the runtime comes from the behavior of the annealing schedule itself.

We consider run lengths (in number of SA evaluations) $N = \{2000, 16000, 128000, 1024000\}$. To additionally show the extra time benefit gained by multi-start SA, for each of the run lengths N, we vary the number of restarts. Beginning with the single run case, we also consider a number of restarts $R = \{1, 2, 4, \ldots\}$. The maximum number of restarts that we consider varies based on N, and is such that $R \cdot N = 16384000$. For example, for the longest run length $N = 1024000$, we consider R only up to 16; while for the shortest run length $N = 2000$, we consider number of restarts R up to 8192.

For each combination of N and R, we execute each of the two versions of the annealing schedule 100 times, and compute the average CPU time. We statistically test the difference in the average CPU times using a t-test.

The experimental results are shown in Figure 1. For the longer run lengths ($N = 128000$ and $N = 1024000$ in Figure 1 parts (c) and (d)), all comparisons are extremely statistically significant based on t-tests, even without restarts. The P-values from t-tests for the runs of length $N = 1024000$ range from $P < 10^{-38}$ without restarts (i.e., when $R = 1$) to $P = 10^{-177}$ for $R = 16$ restarts. The P-values from t-tests for the runs of length $N = 128000$ range from $P < 0.005$ without restarts (i.e., when $R = 1$) to $P < 10^{-223}$ for $R = 128$ restarts. For the runs of length $N = 16000$, the CPU time differences are statistically significant beginning at 8 restarts ($P = 0.011$), and the CPU time differences become increasingly statistically significant as number of restarts increases. For the extremely short run length ($N = 2000$ SA iterations), the benefit to using the optimized version doesn't appear until you restart 128 or more times (e.g., $P < 10^{-7}$ for 128 restarts of a 2000 iteration run). With fewer than 128 restarts, the CPU time differences are not statistically significant for the $N = 2000$ iteration runs.

Just how much faster is the optimized version? Well, for runs of length $N = 1024000$, the optimized version of the Modified Lam is approximately 14% to 15% faster than the original, depending upon number of restarts. For runs of length $N = 128000$ iterations, the optimized version is approximately 27% to 35% faster than the original, depending upon number of restarts. For the shorter runs of $N = 16000$ iterations, specifically for the numbers of restarts where a statistically significant difference in run times was observed (when $R \geq 8$), the optimized version was approximately 25% to 32% faster than the original. Likewise, if the number of restarts of the very short runs of length $N = 2000$ was high enough to lead to a statistically significant difference in run times, the optimized version was 25% to 30% faster.

Thus, for longer run lengths, even without restarts, the optimized version exhibits an extremely statistically significant performance advantage over the original. For short run lengths, the optimized version is faster without restarts (but not at statistically significant levels). However, it is unusual to use such short run lengths without restarting. When considering restarts with the short run lengths, the performance advantage of the optimized version is again very statistically significant.

The reason the optimized version has an added benefit when restarts are used is related to the ability to cache the results of the exponentiations. The optimized version only computes two exponentiations across the restarts, whereas the original version computes a number of exponentiations that is linear in the product of the run length N and number of restarts R.

[4] https://chips-n-salsa.cicirello.org/api/

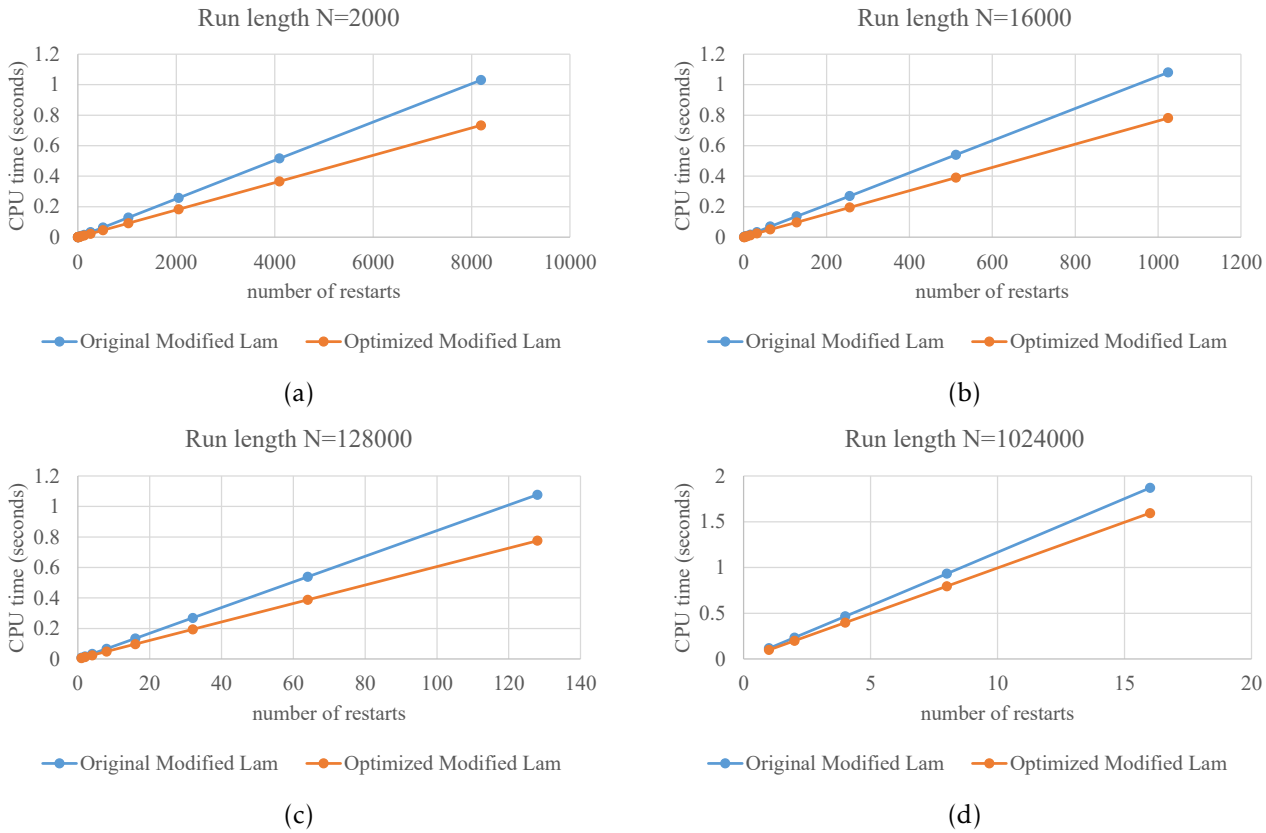

Figure 1. Runtime comparison of the original and optimized versions of the Modified Lam annealing schedule for runs of length: (a) 2000 iterations, (b) 16000 iterations, (c) 128000 iterations, and (d) 1024000 iterations. Each graph shows average CPU time in seconds as a function of number of restarts.

4.3. Effects on Simulated Annealing

In the previous section, which considered the runtime of the annealing schedule independent of an actual run of SA, we saw that the optimized version speeds up the annealing schedule computation by 15% to 30% depending upon run length. However, how much faster the SA actually becomes depends upon the cost of random neighbor generation and cost function computation relative to the cost of the annealing schedule. In this section, we explore this further by considering four different benchmarking problems, each using a different solution representation.

OneMax: The OneMax (Syswerda, 1989) problem is a well-known problem that has been used for benchmarking genetic algorithms for decades. It is an optimization problem over the space of vectors of bits. The problem is to find the vector of bits (of some specified length) that maximizes the number of one-bits. The optimal solution is trivially the vector of all ones.

In our experiments, we use bit vectors of length $L = 20480$. This length was chosen as it is a multiple of 32 bits (although our implementation does not require that), and is just long enough that even our long runs of SA did not optimally solve the instances. The latter is important for the experiments because our SA is designed to terminate in cases where a known optimal cost value is found and we want to ensure that time differences are strictly due to the annealing schedule and not due to early termination. The BitVector class in the Chips-n-Salsa library efficiently implements bit vectors, including methods for accessing individual bits, or groups of bits, as needed, as well as all common bit-wise operators. We consider run lengths $N = \{10^4, 10^5, 10^6\}$ in number of simulated annealing iterations. Our SA implementations are designed to minimize a cost function, so we map the problem of maximizing the number of one bits to minimizing the cost function: L – OneBitCount. Our neighborhood function is "bit flip mutation," commonly used as a mutation operator in genetic algorithms. Our implementation generates a neighbor by picking a random bit position, and flipping its value.

At each run length, we execute the SA 100 times, and report both the average cost of solutions and the average

Table 1. OneMax: (a) optimization cost function, and (b) CPU time. Results are averages of 100 runs, and the P column shows t-test P-values.

	Cost Function Value		
N	Original	Optimized	P
10^4	6422.56	6417.28	0.58
10^5	883.70	888.75	0.25
10^6	12.94	12.56	0.42

(a)

	CPU Time (seconds)		
N	Original	Optimized	P
10^4	0.012	0.012	0.74
10^5	0.106	0.104	0.025
10^6	1.005	0.984	$3.4 \cdot 10^{-38}$

(b)

Table 2. BoundMax: (a) optimization cost function, and (b) CPU time. Results are averages of 100 runs, and the P column shows t-test P-values.

	Cost Function Value		
N	Original	Optimized	P
10^4	553.38	552.11	0.33
10^5	190.90	189.47	0.33
10^6	11.57	11.60	0.95

(a)

	CPU Time (seconds)		
N	Original	Optimized	P
10^4	0.010	0.009	0.38
10^5	0.109	0.106	0.0023
10^6	1.077	1.055	$1.7 \cdot 10^{-39}$

(b)

CPU time of the SA. We include the average cost values to demonstrate that the optimizations do not affect the general behavior of SA, and only affects the runtime. We test statistical significance with t-tests.

Table 1 summarizes the results. In Table 1(a) we can confirm that the difference between the two versions for the average value of the cost function we are minimizing is not statistically significant. This should be the case since we desire our optimized version to be logically equivalent to the original as far as the annealing schedule is concerned. More importantly, however, is that in Table 1(b) we see that the optimized version is faster at very statistically significant levels ($P = 0.025$ for $N = 10^5$ iterations, and P very near 0 for $N = 10^6$ iterations), other than for very short runs where no difference is seen. The optimized version lead to an approximately 2% faster SA than the original version.

BoundMax: We define the BoundMax problem as a generalization of OneMax to searching the space of vectors of integers. Although we are unaware of any prior usages of BoundMax, we do not claim novelty as it is the obvious generalization of OneMax. Specifically, given length L and integer bound $B > 0$, the search space is defined as the set of all vectors V of length L such that the elements V_i are contained within the integer bounds $0 \leq V_i \leq B$. The BoundMax problem is then to maximize the number of elements of V that are equal to B. If the bound $B = 1$ this is just the OneMax problem. We include the BoundMax problem in our experiments to have a simple benchmarking problem with an integer vector representation.

In our experiments, we set the bound $B = 127$ and bit vector length $L = 650$. As in the case of the OneMax experiments, we chose these as they are just right to ensure that even the long SA runs don't quite optimally solve the instances. For the neighborhood function, we use a random value change operator that iterates over the integer vector, and with probability P, changes the element to a different random value within the bounds. We set $P = 1/L$ so that the expected number of integers changed is 1 (it is also implemented to ensure that at least one integer is changed). We convert this maximization problem to a minimization problem in the same way that we did for the OneMax problem. We consider the same run lengths as we did in the OneMax experiments: $N = \{10^4, 10^5, 10^6\}$. And we again report averages of 100 runs for both average solution cost and average CPU time.

Table 2 summarizes the results on the BoundMax problem. Just like in the case of the OneMax problem, we see no difference (statistically) in the cost function values of the solutions to the BoundMax problem produced by SA (Table 2(a)). This is because both the original and optimized versions produce the same sequence of target acceptance rates. However, the optimized version is between 2% and 3% faster (Table 2(b)) at statistically significant levels ($P = 0.0023$ for runs of length $N = 10^5$ SA iterations, and P very near zero for $N = 10^6$ iterations). The time difference for short runs of $N = 10^4$ iterations was not statistically significant.

Permutation in a Haystack: In our prior work, we introduced the Permutation in a Haystack problem (Cicirello, 2016), Haystack(δ, L), as an optimization problem over the space of permutations of the set of integers $\{0, 1, \ldots, (L-1)\}$. The Haystack(δ, L) is to find the permutation p that minimizes the distance $\delta(p, p_L)$ to a target permutation $p_L = [0, 1, \ldots, (L-1)]$. The optimal solution to the problem is trivially p_L. The choice of distance metric $\delta(p, p_L)$ affects the topology of the

Table 3. Permutation in a Haystack: (a) optimization cost function, and (b) CPU time. Results are averages of 100 runs, and the P column shows t-test P-values.

	Cost Function Value		
N	Original	Optimized	P
10^4	775.28	774.73	0.41
10^5	585.71	587.15	0.37
10^6	179.38	179.29	0.96

(a)

	CPU Time (seconds)		
N	Original	Optimized	P
10^4	0.006	0.006	0.67
10^5	0.099	0.098	0.37
10^6	0.939	0.911	$2.3 \cdot 10^{-6}$

(b)

search space, enabling isolating the problem characteristics of interest in your study. There are a large variety of distance metrics on permutations that you can choose from (Cicirello, 2019). One can view the Permutation in a Haystack as an analog of the OneMax problem, but for permutations rather than vectors of bits since maximizing the number of one bits in a vector of bits is equivalent to minimizing the Hamming distance to the vector of all one bits.

In our experiments, we use permutations of length $L = 800$. We use a permutation distance metric known as Exact Match distance for δ. Exact Match distance is essentially Hamming distance, but on permutations, and simply counts the number of positions with different elements. The combination of L and δ were chosen so that even the longest runs of SA don't quite optimally solve the problem. For representing permutations and for computing distance metrics on permutations, we use the Java Permutation Tools (JPT) library (Cicirello, 2018). We use Swap Mutation for the neighborhood function, which picks a pair of random elements and swaps them within the permutation. The run lengths are as defined for the previous problems, $N = \{10^4, 10^5, 10^6\}$, and we average over 100 runs, reporting both average cost values and average CPU times.

The results for the Permutation in a Haystack problem are shown in Table 3. Our results confirm that, as expected, the differences in the optimization cost function values of solutions are not statistically significant (Table 3(a)). Regarding CPU time, as shown in Table 3(b), we only see a statistically significant difference for the longer runs of 10^6 SA iterations (P very near zero) where we find that SA using the optimized version of the Modified Lam is approximately 3% faster than using the original version. The reason that we don't see a statistically significant runtime difference until the longer runs of SA for this problem compared to the others is likely due to the increased time to generate random neighbors of a permutation and to evaluate the optimization cost value for this problem, relative to that of the OneMax and BoundMax problems. That is, although the optimized annealing schedule is 15% to 30% faster as seen in the results of Section 4.2, the time to compute the annealing schedule is a smaller fraction of overall runtime in the case of this permutation optimization problem. So it will likely take much longer runs before we fully realize the performance benefit.

Polynomial Root Finding: Although SA is more commonly used for discrete optimization problems, we wanted to include a case of real-valued function optimization. For this purpose, we use a one-dimensional polynomial root finding problem. The specific instance we used is to find the roots of the polynomial: $12500 - 2500X - 5X^2 + X^3$. This polynomial has three roots: 50, -50, and 5. Therefore, there are three optimal solutions to the problem. We define it as an optimization problem as minimize: $|12500 - 2500X - 5X^2 + X^3|$.

For this problem, we use longer run lengths than we did for the previously because an iteration of SA is much faster for this problem than it was for the previous problems. This is because with this problem, an iteration of SA is operating on a single floating-point value X, representing a solution to the problem; whereas with the OneMax, BoundMax, and Permutation in a Haystack problems, neighbor generation involved operating on a long sequence. Likewise, computing the cost function takes much less time for the Polynomial Root Finding problem since it is just a small number of arithmetic operations; whereas with the other problems cost evaluation involved iterating over a much longer sequence. The run lengths that we use for the Polynomial Root Finding problem are: $N = \{10^5, 10^6, 10^7, 10^8\}$. At each run length, we solve 100 times and report the average cost value and average CPU time.

The results for the Polynomial Root Finding problem are found in Table 4. As in all of the prior problems, there is no difference statistically in the cost of solutions found by the two versions (Table 4(a)), as we should expect since the annealing schedule consists in the same sequence of target acceptance rates regardless of the approach to computing it. The savings in CPU time, however, is much more substantial for the Polynomial Root Finding problem than it was for any of the other problems (Table 4(b)). All run lengths, even the shortest run length, exhibited statistically significant differences in CPU time (P values were very near zero). For the shortest run length (10^5 SA iterations), the optimized version was 50% faster, which might be due to a

Table 4. Polynomial Root Finding: (a) optimization cost function, and (b) CPU time. Results are averages of 100 runs, and the P column shows t-test P-values.

	Cost Function Value		
N	Original	Optimized	P
10^5	0.00635	0.00589	0.60
10^6	0.00057	0.00061	0.60
10^7	0.00006	0.00005	0.08
10^8	0.00001	0.00001	0.70

(a)

	CPU Time (seconds)		
N	Original	Optimized	P
10^5	0.010	0.005	$3.6 \cdot 10^{-6}$
10^6	0.080	0.062	$3.5 \cdot 10^{-66}$
10^7	0.803	0.614	$5.8 \cdot 10^{-208}$
10^8	8.038	6.144	$3.1 \cdot 10^{-260}$

(b)

measurement anomaly related to measuring very short runtimes, since our earlier results when the annealing schedule was isolated from SA (see Section 4.2) found at most a 30% speedup of the annealing schedule. For the other three run lengths (10^6, 10^7, and 10^8 SA iterations), the SA using the optimized version of the annealing schedule is approximately 22% to 24% faster than the original version. This shows that there is potential extreme time savings in cases where the cost function and neighbor generation are simpler.

5. Conclusions

In this paper, we presented an optimized version of an existing adaptive annealing schedule for SA, known as the Modified Lam. The original version of the Modified Lam annealing schedule computes $O(N)$ exponentiations where N is the run length in number of SA iterations. Our optimized version, computes only two exponentiations at the start, and then $O(N)$ multiplications during the run. Furthermore, in the case of multistart SA, where you run SA multiple times and return the best solution from the restarts, our optimized version has the added advantage that the two exponentiations can be cached so that you only need to compute two exponentiations total across all restarts.

In our experiments, we first showed that the optimized version of the annealing schedule is between 15% and 32% faster than the original depending upon run length and number of restarts. The effects on the overall run time of the SA as a whole is not as extreme since the SA is also computing random neighbors as well as computing the optimization cost function during each iteration. In some cases, the SA as a whole is 2% to 3% faster, such as for representations like integer vectors and bit vectors where random neighbor generation and cost function computation likely dominate the runtime of an iteration relative to updating the annealing schedule. This was even more the case with the permutation optimization problem, where the optimized version was 3% faster but only for the long run lengths. We should expect the time savings in these cases to grow if restarts are used due to the effects of caching the exponentiations. We also saw that there are problems where the optimized version exhibits very substantial time savings of 22% to 24% for problems where generating a random neighbor and evaluating its cost are simpler operations (e.g., the polynomial root finding problem). So although the time savings is sometimes modest for some problems, the optimizations never lead to a slower runtime, and have the potential to provide very large overall time savings.

References

Boyan, J.A. (1998) *Learning Evaluation Functions for Global Optimization*. Ph.D. thesis, Carnegie Mellon University, USA.

Cicirello, V.A. (2016) The permutation in a haystack problem and the calculus of search landscapes. *IEEE Transactions on Evolutionary Computation* **20**(3): 434–446. doi:10.1109/TEVC.2015.2477284.

Cicirello, V.A. (2017) Variable annealing length and parallelism in simulated annealing. In *Proceedings of the Tenth International Symposium on Combinatorial Search (SoCS 2017)* (AAAI Press): 2–10. URL https://www.cicirello.org/publications/SoCS2017-Cicirello.pdf.

Cicirello, V.A. (2018) JavaPermutationTools: A java library of permutation distance metrics. *Journal of Open Source Software* **3**(31). doi:10.21105/joss.00950.

Cicirello, V.A. (2019) Classification of permutation distance metrics for fitness landscape analysis. In *Proceedings of the 11th International Conference on Bio-inspired Information and Communication Technologies* (Springer Nature): 81–97. doi:10.1007/978-3-030-24202-2_7.

Cicirello, V.A. (2020) Chips-n-salsa: A java library of customizable, hybridizable, iterative, parallel, stochastic, and self-adaptive local search algorithms. *Journal of Open Source Software* **5**(52). doi:10.21105/joss.02448.

Cismaru, D.C. (2018) Energy efficient train operation using simulated annealing algorithm and simulink model. In *2018 International Conference on Applied and Theoretical Electricity (ICATE)*: 1–4. doi:10.1109/ICATE.2018.8551415.

Daryanavard, H. and Harifi, A. (2019) Uav path planning for data gathering of iot nodes: Ant colony or simulated annealing optimization. In *2019 3rd International Conference on Internet of Things and Applications (IoT)*: 1–4. doi:10.1109/IICITA.2019.8808834.

Delahaye, D., Chaimatanan, S. and Mongeau, M. (2019) Simulated annealing: From basics to applications. In Gendreau, M. and Potvin, J.Y. [eds.] *Handbook of Metaheuristics* (Springer), 1–35. doi:10.1007/978-3-319-91086-4_1.

Dinh, M.H., Nguyen, V.D., Truong, V.L., Do, P.T., Phan, T.T. and Nguyen, D.N. (2019) Simulated annealing

for the assembly line balancing problem in the garment industry. In *Proceedings of the Tenth International Symposium on Information and Communication Technology* (Association for Computing Machinery): 36–42. doi:10.1145/3368926.3369698.

Hinterding, R. (1995) Gaussian mutation and self-adaption for numeric genetic algorithms. In *Proceedings of 1995 IEEE International Conference on Evolutionary Computation*, **1**: 384–389. doi:10.1109/ICEC.1995.489178.

Hu, C., Duan, Q., Hu, L., Lu, P., Li, Z., Yang, M., Wang, J. *et al.* (2019) An analytical-based hybrid algorithm for fpga placement. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI* (Association for Computing Machinery): 351–354. doi:10.1145/3299874.3318035.

Hubin, A. (2019) An adaptive simulated annealing em algorithm for inference on non-homogeneous hidden markov models. In *Proceedings of the International Conference on Artificial Intelligence, Information Processing and Cloud Computing* (ACM Press): 1–9. doi:10.1145/3371425.3371641.

Jha, S. and Menon, V. (2014) Bbmttp: Beat-based parallel simulated annealing algorithm on gpgpus for the mirrored traveling tournament problem. In *Proceedings of the High Performance Computing Symposium* (Society for Computer Simulation International): 3:1–3:7.

Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P. (1983) Optimization by simulated annealing. *Science* **220**(4598): 671–680. doi:10.1126/science.220.4598.671.

Lam, J. and Delosme, J.M. (1988) Performance of a new annealing schedule. In *Proceedings of the 25th ACM/IEEE Design Automation Conference* (IEEE Computer Society Press): 306–311. doi:10.1109/DAC.1988.14775.

Li, J., Li, L., Yu, F., Ju, Y. and Ren, J. (2019) Application of simulated annealing particle swarm optimization in underwater acoustic positioning optimization. In *OCEANS 2019*: 1–4. doi:10.1109/OCEANSE.2019.8867063.

Liang, Y., Gao, S., Wu, T., Wang, S. and Wu, Y. (2018) Optimizing bus stop spacing using the simulated annealing algorithm with spatial interaction coverage model. In *Proceedings of the 11th ACM SIGSPATIAL International Workshop on Computational Transportation Science* (Association for Computing Machinery): 53–59. doi:10.1145/3283207.3283212.

Ludwin, A. and Betz, V. (2011) Efficient and deterministic parallel placement for fpgas. *ACM Transactions on Design Automation of Electronic Systems* **16**(3): 22:1–22:23. doi:10.1145/1970353.1970355.

Ma, B., He, Y., Du, J. and Han, M. (2019) Research on path planning problem of optical fiber transmission network based on simulated annealing algorithm. In *2019 IEEE 8th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*: 1298–1301. doi:10.1109/ITAIC.2019.8785544.

Rahimian, F., Payberah, A.H., Girdzijauskas, S., Jelasity, M. and Haridi, S. (2015) A distributed algorithm for large-scale graph partitioning. *ACM Transactions on Autonomous and Adaptive Systems* **10**(2): 12:1–12:24. doi:10.1145/2714568.

Ram, D.J., Sreenivas, T.H. and Subramaniam, K.G. (1996) Parallel simulated annealing algorithms. *Journal of Parallel and Distributed Computing* **37**: 207—-212. doi:10.1006/jpdc.1996.0121.

Rudolph, G. (1993) Massively parallel simulated annealing and its relation to evolutionary algorithms. *Evolutionary Computation* **1**(4): 361–383. doi:10.1162/evco.1993.1.4.361.

Sun, W. and Zhang, L. (2018) Wsn location algorithm based on simulated annealing co-linearity dv-hop. In *2018 2nd IEEE Advanced Information Management,Communicates,Electronic and Automation Control Conference (IMCEC)*: 1518–1522. doi:10.1109/IMCEC.2018.8469558.

Swartz, W.P. (1993) *Automatic Layout of Analog and Digital Mixed Macro/Standard Cell Integrated Circuits*. Ph.D. thesis, Yale University.

Syswerda, G. (1989) Uniform crossover in genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms* (Morgan Kaufmann Publishers Inc.): 2–9.

Štefankovič, D., Vempala, S. and Vigoda, E. (2009) Adaptive simulated annealing: A near-optimal connection between sampling and counting. *Journal of the ACM* **56**(3): 18:1–18:36. doi:10.1145/1516512.1516520.

Yan, L., Hu, W. and Han, L. (2019) Optimize spl test cases with adaptive simulated annealing genetic algorithm. In *Proceedings of the ACM Turing Celebration Conference* (Association for Computing Machinery): 1–7. doi:10.1145/3321408.3326676.

Zamli, K.Z., Safieny, N. and Din, F. (2018) Hybrid test redundancy reduction strategy based on global neighborhood algorithm and simulated annealing. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications* (Association for Computing Machinery): 87–91. doi:10.1145/3185089.3185146.

Zhichao, Z., Yuhong, D., Yuqin, D., Jintian, Y. and Renjie, L. (2018) A simulated annealing white balance algorithm for foreign fiber detection. In *Proceedings of the 2nd International Conference on Biomedical Engineering and Bioinformatics* (Association for Computing Machinery): 160–164. doi:10.1145/3278198.3278214.