

RESEARCH ARTICLE

Parallel late acceptance hill-climbing for binary-encoded optimization problems

Emrullah Sonuç^{1,2*} and Ender Özcan²
¹Department of Computer Engineering, Karabuk University, Türkiye

²Computational Optimisation and Learning (COL) Lab, School of Computer Science, University of Nottingham, United Kingdom
esonuc@karabuk.edu.tr, ender.ozcan@nottingham.ac.uk

ARTICLE INFO

Article History:

Received: September 30, 2024

Accepted: February 7, 2025

Published Online: April 7, 2025

Keywords:

Late acceptance hill-climbing

Max-cut problem

Metaheuristics

Uncapacitated facility location problem

Parallel algorithms

AMS Classification:

68T20; 90C27

ABSTRACT

This paper presents a Parallel Late Acceptance Hill-Climbing (PLAHC) algorithm for solving binary-encoded optimization problems, with a focus on the Uncapacitated Facility Location Problem (UFLP) and the Maximum Cut Problem (MCP). The experimental results on various benchmark problem instances demonstrate that PLAHC significantly improves upon the sequential implementation of the standard Late Acceptance Hill-Climbing method in terms of solution quality and computational efficiency. For UFLP instances, an 8-thread implementation with a history list length of 50 achieves the best results, while for MCP instances, a 4-thread implementation with a history list length of 100 is the most effective configuration. The speedup analysis shows performance improvements ranging from 3.33x to 10.00x for UFLP and 2.72x to 9.20x for MCP as the number of threads increases. The performance comparisons to the state-of-the-art algorithms illustrate that PLAHC is highly competitive, often outperforming existing sequential methods, indicating the potential of exploiting parallelism to improve heuristic search algorithms for complex optimization problems.



1. Introduction

A Binary-Encoded Optimization Problem (BEOP) is a type of optimization problem in which the solution is represented as a sequence of binary values (0s and 1s). BEOPs are used in many different areas of operations research, including knapsack problems,^{1,2} feature selection³ maximum cut problem⁴ and facility location problem.⁵ Due to the nature of BEOPs, the exponential growth of the number of variables in the search space makes the process of performing a comprehensive search inapplicable for complex problems. For this purpose, the design of heuristic/metaheuristic algorithm for these problems is the main motivation of researchers.⁶

Hill-Climbing (HC) is one of the simplest heuristic method to handle BEOPs and similar

problems. While HC accepts a non-worse solutions during the search process, it sticks a local optimum which prevents the diversification. To tackle this, several variants of HC algorithm have been proposed such as β -HC,⁷ Late Acceptance HC,⁸ and Expansion-based HC.⁹

The Late Acceptance HC (LAHC) algorithm, first proposed in 2012, is a version of the classical HC method designed to address the limitations of early convergence and getting stuck in local optimum.¹⁰ Unlike traditional HC, LAHC does not immediately reject moves that lead to worse solutions. It compares the current solution with several previous solutions instead of the previous one, and then accepts or rejects it. Accepting worse solutions early in the search process allows for a more robust exploration of the solution space and generally provides better convergence

*Corresponding Author

to the global optimum. Experiments on examples of the traveling salesman problem verify the effectiveness of the proposed method and demonstrate its ability to obtain competitive results without the need for any additional parameters.⁸ In addition, LAHC has outperformed some heuristic algorithms in an international competition for solving the magic square problem. The simplicity of the algorithm and its ease of implementation make it a powerful tool for optimization problems.

Although the LAHC algorithm is a useful method for optimization problems, it has some disadvantages. Since a new solution is obtained from a single solution at each step, the method can get stuck at a local optimum if the history list is not long enough. One of the simplest ways to prevent this is to increase the history list length. However, this also requires fine tuning. This study aims to overcome these disadvantages by developing a parallelized version of the LAHC algorithm focusing to solve BEOPs, which is designed specifically for multi-core CPU environments. This is to significantly speed up the search process thanks to parallel computation while preserving the quality of the search results. The main contributions of this paper are summarized as follows.

- We propose a Parallel LAHC (PLAHC) algorithm for solving BEOPs, with a focus on the Uncapacitated Facility Location Problem (UFLP) and Maximum Cut Problem (MCP).
- PLAHC obtain higher quality results in a shorter time compared to the sequential LAHC algorithm. The experiments demonstrate a significant performance improvements through parallelization, with speedups ranging from 3.33x to 10.00x for UFLP instances and 2.72x to 9.20x for MCP instances as the number of threads increases.
- Comprehensive comparison with state-of-the-art algorithms, showing that PLAHC is highly competitive and often outperforms existing methods for both UFLP and MCP.
- Analysis of the trade-offs between parallelism and history list length, providing insights into the balance between exploration and exploitation in the search process.

2. Background

2.1. Late acceptance hill-climbing

LAHC is a stochastic local search to explore the solution space. The algorithm selectively accepts

new solutions that are better than the solution kept in the queue. This is done by maintaining a history list, which is how the algorithm remembers past solutions. By keeping a record of past solutions, the algorithm compares new candidates with known good solutions and explores other possibilities. Figure 1 presents the pseudocode of LAHC.¹¹

Over the years, several versions of the LAHC algorithm have been proposed to improve its performance and applicability. These include adaptive mechanisms for dynamically adjusting parameters, hybrid approaches combining LAHC with other metaheuristics, and parallel implementations for efficient exploration of large solution spaces.¹² In addition, research efforts have focused on optimizing the algorithm’s parameters and fine-tuning its behavior for specific problem domains. Its adaptability to problems such as exam scheduling and the traveling salesman problem was demonstrated, with both its versatility and areas for improvement being highlighted.¹³ A LAHC-based memetic algorithm for selecting features for recognizing facial emotions is presented and tested on multiple datasets.¹⁴

To enhance efficiency in complex scenarios, the parallel version of LAHC was proposed, with the google machine reassignment problem being specifically addressed.¹⁵ Superior performance to single-threaded LAHC was demonstrated by this adaptation, and comparable results to advanced search algorithms were achieved, emphasizing its potential for resource allocation in large-scale computing environments. A specialized modification, custom LAHC was introduced to improve solution quality for traveling salesman problem instances.¹⁶ For anomaly detection, LAHC was combined with genetic programming, resulting in competitive models with fewer parameters being produced.¹⁷

A recent literature indicates that the LAHC algorithm has been demonstrated to be an effective tool for solving a wide range of optimization problems in various application areas.^{18–20} The balance between exploration and exploitation, and the avoidance of local optima, make LAHC a valuable tool for researchers seeking efficient solutions to complex optimization problems. However, the efficacy of the parallel LAHC algorithm in accelerating optimization processes and overcoming large-scale optimization problems remains a topic of ongoing investigation. While parallel implementations of LAHC have been previously proposed, our PLAHC algorithm differs in several key aspects from existing approaches.¹⁵ Other

$$x_{ij} - y_i \leq 0, \quad i = 1, \dots, n, \quad j = 1, \dots, m \quad (3)$$

$$x_{ij} \in \{0, 1\}, \quad i = 1, \dots, n, \quad j = 1, \dots, m \quad (4)$$

$$y_i \in \{0, 1\}, \quad i = 1, \dots, n. \quad (5)$$

In the Eq. 1, c_{ij} represents the cost of servicing demand from facility i for customer j , while f_i denotes the fixed cost associated with opening a facility at location i . The objective of the UFLP is to minimize the total cost, which comprises both service costs and facility opening costs, as represented by the first and second terms in Eq. (1), respectively. Constraint (2) states that each customer must be served by exactly one facility. Constraint (3) ensures that customers cannot receive service from a facility that is not operational. Constraints (4) and (5) define the binary nature of the decision variables, restricting them to values of either 0 or 1.

Exact algorithms for UFLP developed since the 1960s include branch-and-bound,²² improved linear programming methods,^{23,24} and hybrid approaches.²⁵ However, these methods often require large amounts of memory or computational time, which limits their applicability to small-scale problems.²⁶ Exact solvers struggle with larger instances of 100-500 facilities.²⁷ As a result, metaheuristics have become preferred due to their ability to efficiently find near-optimal solutions in larger UFLP instances.²⁸⁻³¹

2.2.2. Maximum cut problem

The Maximum Cut Problem (MCP) is a maximization BEOP in graph theory and combinatorial optimization. An undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. The aim is that partition V into two subsets S and T ($V - S$) such that the total weight (or number) of edges with one endpoint in S and the other in T is maximized. The problem can be encoded as a binary-encoding where each bit represents a vertex. A '0' might indicate the vertex is in set S , while a '1' indicates it's in set T (or vice versa). The mathematical formulation of MCP is as follows:

$$\text{maximize} \quad \sum_{(u,v) \in E} w(u,v) \cdot [x_u \neq x_v] \quad (6)$$

where x_u and x_v represent the binary values assigned to nodes u and v in Eq. 6, respectively. If $x_u \neq x_v$, then these two nodes are in different sets and the edge is cut.

MCP has been addressed by various heuristic and metaheuristic approaches. Recent literature reveals a diverse range of optimization techniques,

including evolutionary algorithms, scatter search, and hybrid methods.³² Randomized heuristics like GRASP and VNS were proposed yielding near-optimal solutions.³³ Another metaheuristic algorithm was proposed to compare the performance against traditional genetic algorithms.³⁴ An advanced scatter search³⁵ and a tabu search based hybrid evolutionary algorithm³⁶ have showed competitive performance in solving MCP. A tabu search algorithm was presented to effectively solve large-scale MCPs.³⁷ Recent focus has shifted to adapting continuous optimization algorithms for discrete problems, introducing the binary evolutionary algorithm (BinBRO)³⁸ and a one-dimensional binary evolutionary algorithm (oBABC).³⁹ Another study is proposed a novel optimization algorithm called fixed set search for solving MCP, demonstrating effectiveness over GRASP by incorporating a learning procedure.⁴⁰ These studies demonstrate the ongoing evolution of optimization approaches for MCP, with hybrid methods and binary adaptations showing promise for future research.

3. The proposed parallel late acceptance hill-climbing

To address the limitations of the LAHC algorithm, such as the computational time required to reach an optimal solution by iteratively searching with a single solution, the algorithm can be split into independent tasks suitable for concurrent execution. Task splitting identifies opportunities for parallelism, such as evaluating candidate solutions and updating the solution history. The implementation of the parallel LAHC algorithm involves the use of a multi-threaded approach, where multiple threads simultaneously execute different instances of the LAHC. Threads can be used to efficiently manage concurrency and resource allocation. Multiple instances of LAHC can be run simultaneously, allowing for a larger exploration and exploitation of potential solutions in different regions of the search space. The proposed Parallel LAHC (PLAHC) is shown in Figure 2. According to the figure, PLAHC initially reads the problem instance from a file. Then, the number of threads to execute the LAHC algorithm in parallel is set. Here, the history list can be a list that can be used by all threads (shared), or it can be a separate list for each thread (local). In our study, each thread maintains its own history list (local). Following these steps, the specified number of threads (nt) are created, and the parallel section is initiated using OpenMP. In this section, each thread executes its own LAHC algorithm. First, each thread starts by generating a

random initial solution. It then creates a history list of length L filled with this initial fitness value. This list acts as a memory of recent solution qualities.

The core of LAHC is its main loop, which runs for a given number of iterations. At each iteration, each thread generates a new candidate solution using random bit flipping. This process takes the current solution, represented as a bit string, and randomly selects one or more bits to flip - changing 0s to 1s or vice versa. This method of solution generation is crucial because it creates small, random changes to the current solution, allowing the algorithm to explore nearby points in the search space while maintaining the overall structure of the solution. The threads continue to generate candidate solutions and attempt to improve the quality of the results until a predetermined stopping criterion (e.g., a specific number of iterations) is met. Upon satisfaction of the stopping criterion, the parallel section is completed. The best solution of each thread is stored in a shared array accessible to all threads, and the master thread identifies the best solution among them and displays it as the global best solution.

We use multi-threaded programming with OpenMP to speed up the LAHC. Furthermore, different numbers of threads (4, 8, 16 and 32) are tested and the effect on the overall performance of the proposed approach is analyzed.

4. Computational experiments and results

We perform all experiments on a PC with an eight-core processor (Intel(R) Xeon(R) E5-2630 CPU @ 2.40 GHz), with 64 GB of RAM, running the Windows 10 64-bit operating system. PLAHC is implemented in standard C using the gcc compiler with OpenMP version 4.5.

To systematically tune effective parameter settings for PLAHC, we performed different parameter configurations on a subset of representative problem instances. We explore the length of the history list ($L \in \{10, 20, 50, 100\}$) and the number of threads ($nt \in \{4, 8, 16, 32\}$). A fixed computational budget of 80,000 objective function evaluations was allocated for each parameter combination, with 30 independent runs performed to ensure statistical significance.

4.1. Computational results for UFLP

We tested PLAHC on UFLP using the ORLib benchmark set.⁴¹ This problem set has 15 UFLP problems in 4 groups. Table 1 shows the main characteristics of these problems, including the

number of facilities, customers, and optimal solutions. The problem size is denoted as $n \times m$, where n represents the number of facilities and m represents the number of customers.

Table 1. ORLib problem instances for UFLP

Instance	Size	Optimal cost
Cap71	16 x 50	932,615.75
Cap72	16 x 50	977,799.40
Cap73	16 x 50	1,010,641.45
Cap74	16 x 50	1,034,976.98
Cap101	25 x 50	796,648.44
Cap102	25 x 50	854,704.20
Cap103	25 x 50	893,782.11
Cap104	25 x 50	928,941.75
Cap131	50 x 50	793,439.56
Cap132	50 x 50	851,495.33
Cap133	50 x 50	893,076.71
Cap134	50 x 50	928,941.75
CapA	100 x 1000	17,156,454.48
CapB	100 x 1000	12,979,071.58
CapC	100 x 1000	11,505,594.33

First, we test different history list lengths (L) to demonstrate the performance of sequential LAHC on UFLP instances in ORLib. Note that the termination criterion is a predetermined number of function evaluations = 80,000. Each experiment is run for 30 trials for each problem instance. Table 2 shows the sequential LAHC results using the gap score, which is the ratio of the mean to the distance between the mean and the optimum in percent. The results show a clear trend of improved performance as L increases from 10 to 100. The average gap score consistently decreases, with $L = 100$ yielding the best overall results at 0.84. Smaller problem instances (cap71-cap74) perform well across all L values, often reaching optimal solutions, while larger instances benefit more significantly from increasing L values. This is particularly evident for the extra large instances (capa, capb, capc), where significant decreases in gap scores are observed.

Table 3 shows the gap scores of PLAHC using 4 threads. The best overall performance is achieved with $L = 50$, with an average gap score of 0.35, closely followed by $L = 100$ with 0.44. Small and medium instances consistently achieve optimal or near-optimal solutions, even with shorter history lists. Larger instances still benefit from larger L values, with $L = 100$ generally performing best for complex problems. For the largest instances (capa, capb, capc), $L = 50$

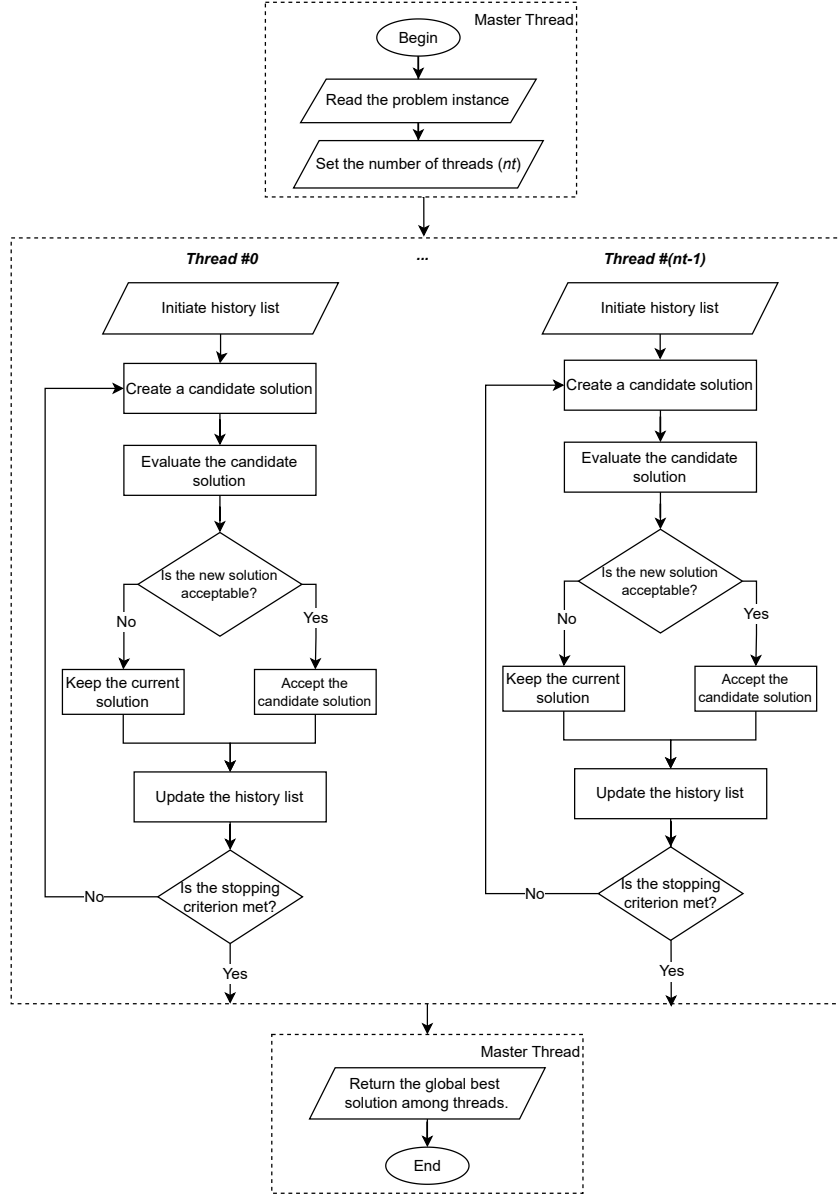


Figure 2. The flowchart of the proposed PLAHC algorithm

outperforms $L = 100$, suggesting that parallelization allows for more efficient exploration of the solution space with shorter history lists. This parallel approach appears to mitigate the need for very long history lists in some cases, likely due to multiple threads simultaneously exploring different regions of the solution space. Overall, the 4-thread implementation significantly improves the effectiveness of the algorithm, achieving better results with shorter history lists than the sequential version, potentially reducing computational requirements while improving solution quality.

Table 4 shows the gap scores of PLAHC using 8 threads. It shows further performance improvements over the 4-thread version. It achieves the best overall results at $L = 50$, with an average gap score of 0.29. Small and medium instances consistently reach optimal solutions over most L values, while larger instances benefit from

increasing L . In particular, $L = 50$ significantly outperforms other configurations for the largest instances, suggesting that increased parallelism allows efficient solution space exploration with moderate history list lengths. However, an unexpected performance drop occurs at $L = 100$, especially for the largest problems, indicating that very long history lists combined with high parallelism can sometimes lead to suboptimal exploration. This 8-thread implementation highlights the improved effectiveness of the algorithm with increased parallelism, achieving superior results with shorter history lists.

Table 5 shows the gap scores of PLAHC using 16 threads. It achieves optimal solutions for small and medium instances across all history list lengths, demonstrating good performance. However, its performance on the largest

Table 2. Gap scores of sequential LAHC on UFLP instances for different L values

Instance	L=10	L=20	L=50	L=100
cap71	0.03	0.06	0.00	0.00
cap72	0.23	0.00	0.00	0.00
cap73	0.17	0.00	0.05	0.01
cap74	0.00	0.05	0.00	0.00
cap101	0.15	0.09	0.10	0.03
cap102	0.13	0.14	0.08	0.02
cap103	0.25	0.09	0.06	0.05
cap104	0.73	0.18	0.07	0.00
cap131	0.81	0.46	0.33	0.35
cap132	0.42	0.36	0.15	0.22
cap133	0.57	0.47	0.20	0.17
cap134	0.44	0.54	0.28	0.12
capa	7.92	7.78	6.05	5.90
capb	5.53	4.17	3.28	3.16
capc	4.49	3.69	3.38	2.64
<i>Avg.Gap</i>	<i>1.46</i>	<i>1.21</i>	<i>0.94</i>	<i>0.84</i>

Table 3. Gap scores on UFLP instances for different values of L on using 4 threads

Instance	L=10	L=20	L=50	L=100
cap71	0.00	0.00	0.00	0.00
cap72	0.00	0.00	0.00	0.00
cap73	0.10	0.00	0.00	0.00
cap74	0.00	0.00	0.00	0.00
cap101	0.06	0.00	0.00	0.00
cap102	0.05	0.00	0.00	0.00
cap103	0.05	0.05	0.02	0.00
cap104	0.00	0.00	0.00	0.00
cap131	0.35	0.26	0.16	0.10
cap132	0.14	0.18	0.03	0.01
cap133	0.15	0.12	0.09	0.02
cap134	0.02	0.00	0.00	0.00
capa	5.14	2.56	1.45	2.89
capb	2.67	2.72	2.32	2.09
capc	3.01	2.33	1.16	1.48
<i>Avg.Gap</i>	<i>0.78</i>	<i>0.55</i>	<i>0.35</i>	<i>0.44</i>

Table 4. Gap scores on UFLP instances for different values of L on using 8 threads

Instance	L=10	L=20	L=50	L=100
cap71	0.00	0.00	0.00	0.00
cap72	0.00	0.00	0.00	0.00
cap73	0.00	0.00	0.00	0.00
cap74	0.00	0.00	0.00	0.00
cap101	0.04	0.00	0.00	0.00
cap102	0.02	0.00	0.00	0.00
cap103	0.05	0.00	0.00	0.00
cap104	0.00	0.00	0.00	0.00
cap131	0.21	0.31	0.06	0.05
cap132	0.01	0.05	0.00	0.01
cap133	0.05	0.05	0.04	0.01
cap134	0.01	0.00	0.00	0.00
capa	3.94	2.52	1.30	6.39
capb	2.47	2.26	1.82	2.61
capc	2.35	1.74	1.16	1.91
<i>Avg.Gap</i>	<i>0.61</i>	<i>0.46</i>	<i>0.29</i>	<i>0.73</i>

instances (capa, capb, capc) decreases as the history list length increases, especially at $L=100$. This leads to a significant increase in the average gap score for longer history lists, contrary to previous trends. The best overall performance is achieved with $L = 20$ with an average gap of 0.34, closely followed by $L = 10$ with an average gap of 0.45, suggesting that shorter history lists are more effective with high parallelism. This behavior indicates that excessive parallelism combined with longer history lists may cause over-exploration or thread interference, resulting in suboptimal solutions for extra large problems.

Table 6 shows the gap scores of PLAHC using 32 threads. While maintaining optimal solutions for small and medium instances across all history list lengths, it shows a dramatic deterioration in performance for larger instances as the history list length increases. The best overall performance is achieved with the shortest history lists: $L = 10$ with an average gap of 0.32 and $L = 20$ with an average gap of 0.33. Performance degrades significantly for $L = 50$ and $L = 100$, primarily due to poor results on the largest instances. These results indicate that as parallelism increases to this level, shorter history lists become more effective, likely because high parallelism already provides diverse exploration. This implementation highlights the need to adjust the tradeoff between parallelism and history length for optimal performance in highly parallel environments, encouraging shorter history lists as the number of threads increases.

The comparison of all experimental results shows a clear performance improvement through parallelization over the sequential implementation. The 8-thread parallel implementation with $L = 50$ emerges as the best overall configuration, achieving the lowest average gap score of 0.29. This configuration provides an optimal balance between parallelism and history list length, and performs well across different problem sizes. As the number of threads increases, the ideal history list length generally decreases, with 16- and 32-thread implementations favoring shorter lists ($L = 20$ and $L = 10$, respectively). However, these high thread counts have shown poor performance for longer history lists, especially for larger problem instances.

Figure 3 shows average speedups for UFLP instances. The results show consistent performance improvements as the number of threads increases. Starting with a 3.33x speedup at 4 threads, performance scales up to 5.52x at 8 threads, 6.96x at 16 threads, and reaches a maximum of 10.00x at 32 threads. Although the speedup is significant, it

exhibits sub-linear scaling, meaning that the efficiency gains decrease as the number of threads increases. This is typical in parallel computing due to factors such as communication overhead and algorithm parts that can't be perfectly parallelized. Despite these limitations, the continued improvement even at 32 threads suggests that high levels of parallelism are beneficial for these UFLP instances. The most significant performance improvements are seen in the initial increments, especially from 4 to 8 threads. Overall, these results suggest that UFLP and PLAHC are well suited for parallelization, although the optimal number of threads depends on specific hardware capabilities and the trade-off between computation time and resource utilization.

The comparative results of the PLAHC (8-thread with $L = 50$), and the state-of-the-art algorithms on the UFLP instances are reported in Table 7. The performance of PLAHC is evaluated by comparing its gap scores and standard deviation (Std) values with those of reference algorithms, as presented in the individual columns of the table. The results of these methods were taken directly from the relevant studies. The best gap values for each instance are shown in bold. oBABC is a metaheuristic algorithm designed to address the limitations of existing binary variants of the ABC algorithm, particularly in solving binary optimization problems such as UFLP.³⁹ MBVS is a metaheuristic algorithm originally designed for continuous optimization problems and later adapted to solve binary problems by transforming continuous values into binary ones.³¹

The performance comparison of PLAHC against oBABC and MBVS shows PLAHC's effectiveness in solving UFLP across various instance sizes. For small to medium instances (cap71-cap104), all three algorithms achieve optimal solutions, with PLAHC matching or slightly outperforming the others. PLAHC's strength becomes more apparent for larger instances (cap131-cap134), where it consistently outperforms or matches the other algorithms. For extra large instances, PLAHC significantly outperforms both oBABC and MBVS on the capa instance with a gap score of 1.2982 compared to 2.8969 and 5.8962 respectively. However, for capb and capc, MBVS outperforms PLAHC. Nevertheless, PLAHC consistently outperforms oBABC on these large instances. In addition, PLAHC generally has lower std values, especially for larger instances, indicating more consistent performance over multiple runs. This combination of competitive solution quality in most instances, especially for complex

Table 5. Gap scores on UFLP instances for different values of L on using 16 threads

Instance	L=10	L=20	L=50	L=100
cap71	0.00	0.00	0.00	0.00
cap72	0.00	0.00	0.00	0.00
cap73	0.00	0.00	0.00	0.00
cap74	0.00	0.00	0.00	0.00
cap101	0.00	0.00	0.00	0.00
cap102	0.00	0.00	0.00	0.00
cap103	0.01	0.00	0.00	0.00
cap104	0.00	0.00	0.00	0.00
cap131	0.16	0.08	0.00	0.04
cap132	0.03	0.00	0.00	0.01
cap133	0.01	0.03	0.05	0.03
cap134	0.00	0.00	0.00	0.00
capa	2.72	1.97	5.73	53.26
capb	2.20	1.57	2.77	20.47
capc	1.66	1.38	2.03	15.10
<i>Avg.Gap</i>	<i>0.45</i>	<i>0.34</i>	<i>0.71</i>	<i>5.93</i>

Table 6. Gap scores on UFLP instances for different values of L on using 32 threads

Instance	L=10	L=20	L=50	L=100
cap71	0.00	0.00	0.00	0.00
cap72	0.00	0.00	0.00	0.00
cap73	0.00	0.00	0.00	0.00
cap74	0.00	0.00	0.00	0.00
cap101	0.00	0.00	0.00	0.00
cap102	0.00	0.00	0.00	0.00
cap103	0.00	0.00	0.00	0.00
cap104	0.00	0.00	0.00	0.00
cap131	0.02	0.06	0.05	1.22
cap132	0.00	0.00	0.01	1.54
cap133	0.00	0.00	0.04	0.75
cap134	0.00	0.00	0.00	2.98
capa	1.76	1.90	52.37	103.31
capb	1.58	1.63	21.14	42.93
capc	1.43	1.36	14.85	31.94
<i>Avg.Gap</i>	<i>0.32</i>	<i>0.33</i>	<i>5.90</i>	<i>12.31</i>

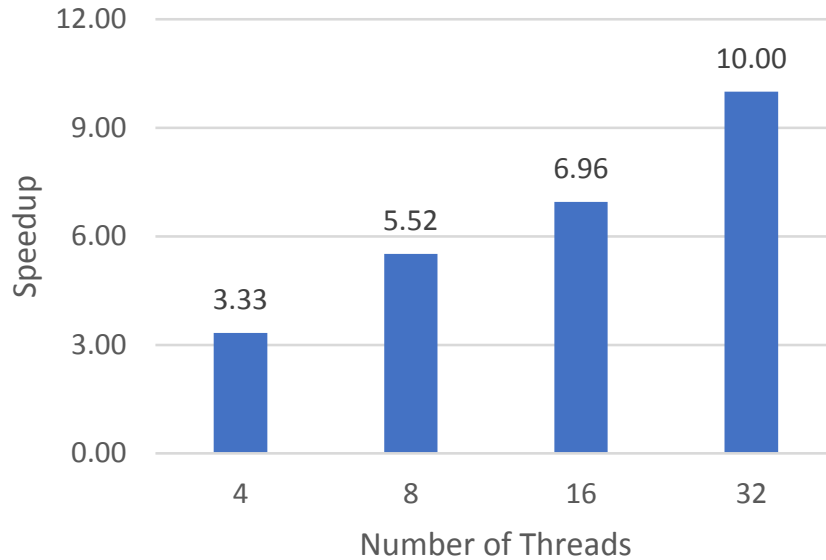
**Figure 3.** Comparison between sequential and parallel implementation of LAHC in terms of average speedup for UFLP instances

Table 7. A comparison with the state-of-the-art algorithms on UFLP instances

Instance	oBABC		MBVS		PLAHC	
	Gap	Std	Gap	Std	Gap	Std
cap71	0.0000	0.00	0.0000	0.00	0.0000	0.00
cap72	0.0000	0.00	0.0000	0.00	0.0000	0.00
cap73	0.0000	0.00	0.0000	0.00	0.0000	0.00
cap74	0.0000	0.00	0.0000	0.00	0.0000	0.00
cap101	0.0000	0.00	0.0000	0.00	0.0000	0.00
cap102	0.0000	0.00	0.0000	0.00	0.0000	0.00
cap103	0.0017	57.34	0.0017	57.34	0.0000	0.00
cap104	0.0000	0.00	0.0000	0.00	0.0000	0.00
cap131	0.1010	813.74	0.0112	270.21	0.0578	691.20
cap132	0.0125	236.72	0.0000	0.00	0.0041	69.92
cap133	0.0409	395.63	0.0591	328.65	0.0416	325.20
cap134	0.0000	0.00	0.0000	0.00	0.0000	0.00
capa	2.8969	299,727.30	5.8962	1,334,986.34	1.2982	96,479.58
capb	2.3595	102,565.90	1.5752	313,844.03	1.8239	89,884.50
capc	2.3283	94,785.55	0.6960	69,018.86	1.1600	30,122.69
<i>Avg.Scores</i>	<i>0.5161</i>	<i>33,238.81</i>	<i>0.5493</i>	<i>114,567.02</i>	<i>0.2924</i>	<i>14,504.87</i>

Table 8. Problem instances of MCP

Instance	Optimal	Instance	Optimal
pw01–100.0	2019	pw05–100.5	8169
pw01–100.1	2060	pw05–100.6	8217
pw01–100.2	2032	pw05–100.7	8249
pw01–100.3	2067	pw05–100.8	8199
pw01–100.4	2039	pw05–100.9	8099
pw01–100.5	2108	pw09–100.0	13585
pw01–100.6	2032	pw09–100.1	13417
pw01–100.7	2074	pw09–100.2	13461
pw01–100.8	2022	pw09–100.3	13656
pw01–100.9	2005	pw09–100.4	13514
pw05–100.0	8190	pw09–100.5	13574
pw05–100.1	8045	pw09–100.6	13640
pw05–100.2	8039	pw09–100.7	13501
pw05–100.3	8139	pw09–100.8	13593
pw05–100.4	8125	pw09–100.9	13658

problems, establishes PLAHC as an effective algorithm for solving the UFLP. It shows remarkable advantages in many instances, although MBVS outperforms it in some larger instances.

4.2. Computational results for MCP

The performance of PLAHC in solving MCP is tested using the benchmark set shown in Table 8. This problem set has 30 MCP problems in 3 groups.⁴² The problem size is 100 for all instances, while the density varies between 0.1, 0.5, and 0.9 for each group, respectively.

We first test different history list lengths (L) to demonstrate the performance of sequential

LAHC on MCP instances. Note that the termination criterion is a predetermined number of function evaluations = 80,000. Table 9 shows the average gap scores of LAHC for each instance, averaged from 30 runs. The results show a clear trend of improvement as L increases from 10 to 100. This improvement is clearly seen for the pw09 instances, where the algorithm’s performance improves significantly at higher L values. In general, $L = 100$ emerges as the best performing configuration, consistently yielding the lowest gap scores for most instances. This suggests that longer L values allow the LAHC algorithm to explore the solution space more effectively, leading to better quality solutions.

Table 9. Gap scores of sequential LAHC on MCP instances for different L values

Instance	L=10	L=20	L=50	L=100
pw01_100.0	2.1313	1.9034	1.8623	1.1823
pw01_100.1	2.9369	1.6311	1.4044	0.2461
pw01_100.2	2.6476	2.0832	1.0138	0.9365
pw01_100.3	1.9434	1.8515	0.9516	1.1403
pw01_100.4	2.6891	1.8293	1.2423	1.5758
pw01_100.5	3.5878	1.9511	2.6409	1.2225
pw01_100.6	2.1491	2.0177	1.4090	2.1196
pw01_100.7	2.9074	1.3679	1.1427	1.1172
pw01_100.8	3.9975	2.3393	1.6632	2.0213
pw01_100.9	2.1032	1.9766	1.8389	1.1920
pw05_100.0	0.9667	0.8165	0.5186	0.3040
pw05_100.1	1.0470	0.9654	0.7201	0.5395
pw05_100.2	0.9715	0.7211	0.7256	0.5424
pw05_100.3	1.1562	0.9620	0.7240	0.4825
pw05_100.4	0.9567	0.8545	0.4538	0.3065
pw05_100.5	1.7371	1.1033	0.7745	0.4587
pw05_100.6	0.8957	0.7724	0.4357	0.3630
pw05_100.7	1.0672	0.7803	0.4659	0.3512
pw05_100.8	1.7742	1.0627	0.7083	0.3607
pw05_100.9	0.9437	1.0532	0.5930	0.3013
pw09_100.0	0.6976	0.4750	0.3340	0.2088
pw09_100.1	0.6162	0.4852	0.3237	0.2556
pw09_100.2	0.8040	0.5416	0.5269	0.3442
pw09_100.3	0.6915	0.6852	0.3603	0.2920
pw09_100.4	0.7543	0.6208	0.4042	0.3295
pw09_100.5	0.6412	0.4938	0.2112	0.2271
pw09_100.6	0.7847	0.6349	0.5320	0.3375
pw09_100.7	0.5220	0.4822	0.4646	0.2889
pw09_100.8	0.6553	0.5481	0.3607	0.2170
pw09_100.9	0.7119	0.7256	0.3385	0.1491
Avg.Gap	1.5163	1.1245	0.8382	0.6471

Table 10 presents the average gap scores on 30 trials with an $L = 50$ of PLAHC using different numbers of threads, which are 4, 8, 16, and 32 shown in the columns. The results show interesting patterns across different problem sizes. For all instances, the 4-thread implementation consistently produces the best results. The gap scores increases as the number of threads increases, suggesting that for these simpler problems, a lower degree of parallelization is most effective. This may be because the overhead of managing more threads outweighs the benefits of increased parallelism for these smaller problem sizes. For the pw09 series, increasing to 16 or 32 threads generally results in a decrease in performance. In summary, these results highlight the importance of carefully matching the degree of parallelization to the specific problem complexity. For the majority of instances, 4-thread implementation provides better balance between parallelism and efficiency.

Table 11 presents the average gap scores with an $L = 100$ of PLAHC. According to the experimental results, across all three groups (pw01, pw05, and pw09), the 4-thread implementation consistently produces the best results, with performance generally decreasing as the number of

threads increases. This can be clearly seen clearly in all instances, where the difference in performance between 4 threads and 32 threads is quite significant.

Comparing all experiments, including sequential implementations with different history list lengths ($L = 10, 20, 50, 100$) and parallel implementations with $L = 50$ and $L = 100$ using different number of threads (4, 8, 16, 32), the best overall configuration emerges as the parallel implementation with $L = 100$ and 4 threads. This configuration consistently produces the best average gaps scores across all problem instance groups (pw01, pw05, pw09). The longer history list ($L = 100$) generally outperforms the shorter ones, indicating the benefits of a larger search history for the sequential LAHC. Parallelization of the LAHC algorithm using 4 threads shows the highest level of efficiency. This is because the overhead associated with parallel processing reduces the benefits of parallelism as the number of threads increases. This optimal configuration successfully balances exploration of the solution space with computational efficiency, demonstrating robustness to varying problem characteristics. It effectively combines the benefits of extensive search history with efficient parallel processing, making it the most appropriate choice

Table 10. Gap scores for MCP instances with different number of threads on PLAHC where $L = 50$

Instance	4 thds	8 thds	16 thds	32 thds
pw01_100.0	0.8504	1.4165	2.4978	4.0451
pw01_100.1	0.3383	0.9141	1.9694	3.3723
pw01_100.2	0.7859	1.3844	1.8209	2.9444
pw01_100.3	0.6918	0.8403	2.7949	4.4881
pw01_100.4	0.9725	2.3409	3.4036	4.7048
pw01_100.5	0.7386	1.7362	3.0773	4.4639
pw01_100.6	0.9070	1.2879	2.7347	4.6344
pw01_100.7	0.5868	1.1172	1.9012	3.4730
pw01_100.8	1.0618	1.7557	2.9228	4.2779
pw01_100.9	0.7247	1.0574	2.3277	3.5711
pw05_100.0	0.1966	0.5076	0.7485	1.0546
pw05_100.1	0.3924	0.5395	0.9475	1.2799
pw05_100.2	0.2920	0.4715	0.6291	0.9852
pw05_100.3	0.2924	0.5439	1.2520	1.6836
pw05_100.4	0.2896	0.3967	0.6994	1.0933
pw05_100.5	0.3791	0.5941	0.9728	1.4494
pw05_100.6	0.2601	0.4044	0.7469	1.1777
pw05_100.7	0.2501	0.3621	0.7743	1.2632
pw05_100.8	0.2143	0.5285	1.1676	1.5746
pw05_100.9	0.1824	0.5503	0.9063	1.3800
pw09_100.0	0.1877	0.4183	0.5067	0.7528
pw09_100.1	0.2253	0.2849	0.4593	0.6449
pw09_100.2	0.3605	0.4546	0.5861	0.7050
pw09_100.3	0.3048	0.4945	0.6178	0.8836
pw09_100.4	0.2573	0.4179	0.6527	0.8448
pw09_100.5	0.1490	0.2981	0.4126	0.5844
pw09_100.6	0.3268	0.5086	0.7036	0.9122
pw09_100.7	0.1701	0.3523	0.4716	0.5994
pw09_100.8	0.1962	0.3995	0.6001	0.7494
pw09_100.9	0.2473	0.4127	0.6094	0.8271
<i>Avg. Gap</i>	<i>0.4277</i>	<i>0.7597</i>	<i>1.3305</i>	<i>2.0140</i>

for the PLAHC algorithm applied to MCP instances.

Figure 4 shows average speedups for MCP instances. Similar to the UFLP experiments, the results show consistent performance improvements as the number of threads increases. Speedup values increase from 2.93x with 4 threads to 11.01x with 32 threads, demonstrating significant improvement through parallelization. However, the observed scaling is non-linear, suggesting that the benefits of adding more threads yield less performance. This can be seen in the decreasing throughput or speedup per thread as the number of threads increases. While 32 threads achieve the highest speedup of 11.01x, it’s important to note that earlier results show that 4-thread implementation often produces the best solution quality. This highlights a critical trade-off in parallel optimization algorithms between speed and solution quality.

The comparative results of PLAHC (4-thread with $L = 100$), and the state-of-the-art algorithms on the MCP instances are reported in Table 12. The performance evaluation of PLAHC is conducted by comparing its average gap and

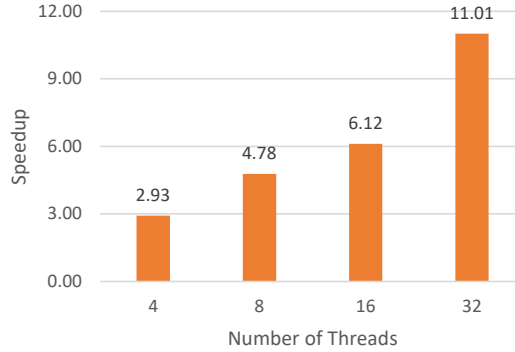
std values. The results of oBABC were taken directly from the the reference study. Note that the termination criterion is a predetermined number of function evaluations = 20,000 to fairly compare the performance of oBABC to ABPEA. The best average gap scores for each instance are shown in bold. The performance comparison between PLAHC and oBABC shows that PLAHC generally outperforms oBABC in most problem instances, especially in the pw01 series. PLAHC consistently achieves lower gap scores, with improvements in several instances such as pw01_100.5, pw05_100.1, and pw09_100.6. While PLAHC typically has higher standard deviations, indicating more diverse solutions, its superior average gap scores indicate a more effective exploration of the solution space. The results highlight PLAHC’s effectiveness in solving MCP instances, likely due to its parallel nature and late acceptance strategy.

4.3. Runtime analysis

To provide a comprehensive understanding of the computational efficiency of PLAHC, we analyzed

Table 11. Gap scores for MCP instances with different number of threads on PLABC where $L = 100$

Instance	4 thds	8 thds	16 thds	32 thds
pw01_100.0	0.7098	1.2566	2.3789	3.7593
pw01_100.1	0.5714	0.9772	1.9888	3.1359
pw01_100.2	0.8022	1.0566	1.7879	2.6068
pw01_100.3	0.8016	1.2274	2.6352	4.3493
pw01_100.4	0.8176	1.7474	3.4036	4.4679
pw01_100.5	0.7936	1.7566	2.9331	4.3819
pw01_100.6	0.8253	1.6845	2.8430	4.3356
pw01_100.7	0.4725	1.0882	1.8963	3.3558
pw01_100.8	0.9545	1.6864	2.7829	4.3917
pw01_100.9	0.6050	0.9362	1.9636	3.5112
pw05_100.0	0.2487	0.5137	0.6972	0.9955
pw05_100.1	0.3198	0.5158	0.6865	1.2098
pw05_100.2	0.2347	0.4756	0.6838	0.9084
pw05_100.3	0.3231	0.6594	1.0718	1.6673
pw05_100.4	0.2511	0.4107	0.6810	1.1434
pw05_100.5	0.3774	0.5211	0.9907	1.4355
pw05_100.6	0.2300	0.4710	0.6730	1.0941
pw05_100.7	0.1197	0.3342	0.6756	1.1472
pw05_100.8	0.1699	0.4744	1.0940	1.4786
pw05_100.9	0.1840	0.5948	0.9647	1.3487
pw09_100.0	0.2088	0.3185	0.4750	0.7499
pw09_100.1	0.1824	0.3421	0.4418	0.6266
pw09_100.2	0.2704	0.4601	0.5839	0.6777
pw09_100.3	0.2631	0.4674	0.6175	0.8577
pw09_100.4	0.1663	0.3732	0.6608	0.7868
pw09_100.5	0.1537	0.2399	0.4165	0.5413
pw09_100.6	0.3556	0.4646	0.6820	0.8908
pw09_100.7	0.2150	0.3538	0.4269	0.6229
pw09_100.8	0.1727	0.2906	0.4995	0.7708
pw09_100.9	0.2077	0.3776	0.5482	0.8471
<i>Avg. Gap</i>	<i>0.4003</i>	<i>0.7359</i>	<i>1.2728</i>	<i>1.9365</i>

**Figure 4.** Comparison between sequential and parallel implementation of LAHC in terms of average speedup for MCP instances

the execution times of both sequential and parallel LAHC implementations across all problem instances. Tables 13 and 14 show the average execution times in seconds for UFLP and MCP instances, respectively.

For UFLP instances, the execution times show a clear correlation with problem dimensions. Small instances (16×50 and 25×50) are solved rapidly, with sequential LAHC requiring only 0.12-0.24 seconds and parallel implementations further reducing this

to 0.02-0.08 seconds. Medium-sized instances (50×50) show moderate computation times of 0.22-0.35 seconds for sequential execution. The most major computational demands are observed in the large instances (100×1000), where sequential execution times reach 7.06-7.38 seconds, highlighting the increased complexity of these problem sizes.

The MCP instances show a clear pattern with respect to graph density. Problems with lower

Table 12. A comparison with the state-of-the-art algorithms on UFLP instances

Instance	oBABC		PLAHC	
	Gap	Std	Gap	Std
pw01-100.0	1.6840	9.18	1.3670	9.72
pw01-100.1	1.1845	16.25	1.1553	14.92
pw01-100.2	1.8455	11.07	1.5502	12.86
pw01-100.3	1.2966	11.34	1.5820	16.41
pw01-100.4	2.0745	13.91	1.9225	14.82
pw01-100.5	2.5427	25.47	2.0873	19.02
pw01-100.6	2.1604	11.2	1.8602	15.99
pw01-100.7	1.3549	15.98	1.2150	13.56
pw01-100.8	1.8200	16.38	2.0870	16.8
pw01-100.9	1.2070	9.96	1.0574	7.82
pw05-100.0	0.5043	22.67	0.4786	19.28
pw05-100.1	0.6750	17.54	0.5991	18.92
pw05-100.2	0.6481	22.61	0.6978	19.48
pw05-100.3	0.7863	31.86	0.9399	35.66
pw05-100.4	0.5600	24.93	0.5157	22.5
pw05-100.5	0.5276	28.52	0.8802	22.81
pw05-100.6	0.4211	15.66	0.4211	22.37
pw05-100.7	0.4413	24.94	0.4667	22.77
pw05-100.8	0.6989	38.79	0.7635	39.12
pw05-100.9	0.6964	27.18	0.7161	27.16
pw09-100.0	0.2657	21.71	0.3526	18.22
pw09-100.1	0.3622	26.45	0.3898	18.75
pw09-100.2	0.4858	31.32	0.4702	21.78
pw09-100.3	0.4379	33.6	0.4196	24.54
pw09-100.4	0.4344	22.97	0.4136	16.49
pw09-100.5	0.2380	27.44	0.3330	20.8
pw09-100.6	0.5916	13.18	0.4985	19.73
pw09-100.7	0.3133	23.44	0.3466	22.63
pw09-100.8	0.3171	28.28	0.3524	22.97
pw09-100.9	0.3507	32.71	0.4049	19.78
<i>Avg.Scores</i>	<i>0.8975</i>	<i>21.88</i>	<i>0.8781</i>	<i>19.92</i>

Table 13. Average execution times on UFLP instances

Instance	LAHC	4 thds	8 thds	16 thds	32 thds
cap71_16_50	0.17	0.06	0.04	0.04	0.03
cap72_16_50	0.16	0.05	0.04	0.04	0.03
cap73_16_50	0.12	0.04	0.03	0.03	0.02
cap74_16_50	0.12	0.04	0.03	0.03	0.02
cap101_25_50	0.24	0.08	0.05	0.05	0.03
cap102_25_50	0.22	0.07	0.05	0.05	0.03
cap103_25_50	0.17	0.06	0.04	0.04	0.03
cap104_25_50	0.15	0.05	0.04	0.04	0.03
cap131_50_50	0.35	0.11	0.07	0.06	0.04
cap132_50_50	0.33	0.10	0.08	0.06	0.04
cap133_50_50	0.30	0.10	0.07	0.05	0.04
cap134_50_50	0.22	0.09	0.06	0.05	0.03
capa_100_1000	7.25	2.13	1.27	0.99	0.65
capb_100_1000	7.06	2.12	1.26	1.01	0.65
capc_100_1000	7.38	2.18	1.30	1.01	0.65
<i>Average</i>	<i>1.61</i>	<i>0.48</i>	<i>0.29</i>	<i>0.23</i>	<i>0.16</i>

Table 14. Average execution times on MCP instances

Instance	LAHC	4 thds	8 thds	16 thds	32 thds
pw01_100.0	0.14	0.06	0.05	0.04	0.04
pw01_100.1	0.15	0.06	0.05	0.05	0.04
pw01_100.2	0.15	0.06	0.05	0.05	0.04
pw01_100.3	0.15	0.06	0.05	0.04	0.04
pw01_100.4	0.15	0.06	0.05	0.04	0.04
pw01_100.5	0.15	0.06	0.05	0.05	0.03
pw01_100.6	0.15	0.06	0.05	0.05	0.04
pw01_100.7	0.15	0.06	0.05	0.04	0.04
pw01_100.8	0.15	0.06	0.05	0.05	0.03
pw01_100.9	0.15	0.05	0.05	0.04	0.03
pw05_100.0	0.67	0.28	0.17	0.13	0.08
pw05_100.1	0.67	0.27	0.17	0.13	0.08
pw05_100.2	0.66	0.27	0.17	0.13	0.08
pw05_100.3	0.67	0.27	0.17	0.13	0.08
pw05_100.4	0.67	0.27	0.17	0.13	0.08
pw05_100.5	0.67	0.27	0.17	0.13	0.08
pw05_100.6	0.66	0.27	0.17	0.13	0.08
pw05_100.7	0.66	0.27	0.17	0.13	0.08
pw05_100.8	0.66	0.28	0.17	0.13	0.07
pw05_100.9	0.67	0.28	0.17	0.13	0.08
pw09_100.0	1.67	0.52	0.30	0.23	0.11
pw09_100.1	1.68	0.52	0.30	0.23	0.11
pw09_100.2	1.67	0.52	0.31	0.23	0.11
pw09_100.3	1.68	0.52	0.29	0.23	0.11
pw09_100.4	1.68	0.51	0.30	0.23	0.11
pw09_100.5	1.67	0.52	0.30	0.23	0.11
pw09_100.6	1.68	0.52	0.30	0.23	0.11
pw09_100.7	1.68	0.52	0.30	0.23	0.12
pw09_100.8	1.69	0.52	0.30	0.23	0.11
pw09_100.9	1.68	0.52	0.31	0.22	0.11
<i>Average</i>	<i>0.83</i>	<i>0.28</i>	<i>0.17</i>	<i>0.14</i>	<i>0.08</i>

density (pw01 series, density=0.1) are solved quickly, requiring only 0.14-0.15 seconds sequentially. Medium-density instances (pw05 series, density=0.5) require approximately 0.67 seconds, while high-density instances (pw09 series, density=0.9) need about 1.68 seconds for sequential execution.

Parallel implementation shows significant efficiency gains across both problem types. For UFLP, the average execution time decreases from 1.61 seconds (sequential) to 0.49, 0.29, 0.24, and 0.15 seconds at 4, 8, 16, and 32 threads, respectively. Similarly for MCP, the average time reduces from 0.83 seconds to 0.28, 0.17, 0.14, and 0.08 seconds as the number of threads increases. The performance improvement is most pronounced when moving from sequential to 4

threads and from 4 to 8 threads, with yields decreasing as the number of threads increases.

4.4. Parametric analysis

The experimental results revealed distinct optimal configurations for different problem instances. Figure 5 and 6 show the results of our parameter tuning analysis through heatmaps for UFLP and MCP, respectively, where lighter shading indicates better performance (lower gap scores). The visualization shows distinct optimal configurations for different problem types. For UFLP instances, the combination of $L = 50$ and $nt = 8$ achieves the best performance with an average gap of 0.29. This configuration provides an effective balance between exploration depth through moderate history length and efficient parallel computation. The MCP instances

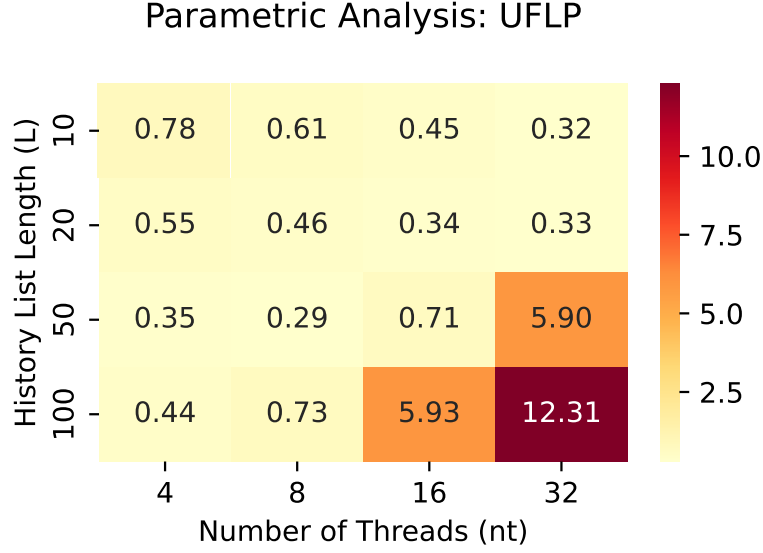


Figure 5. Average gap scores for different history list lengths (L) and number of threads (nt) on UFLP

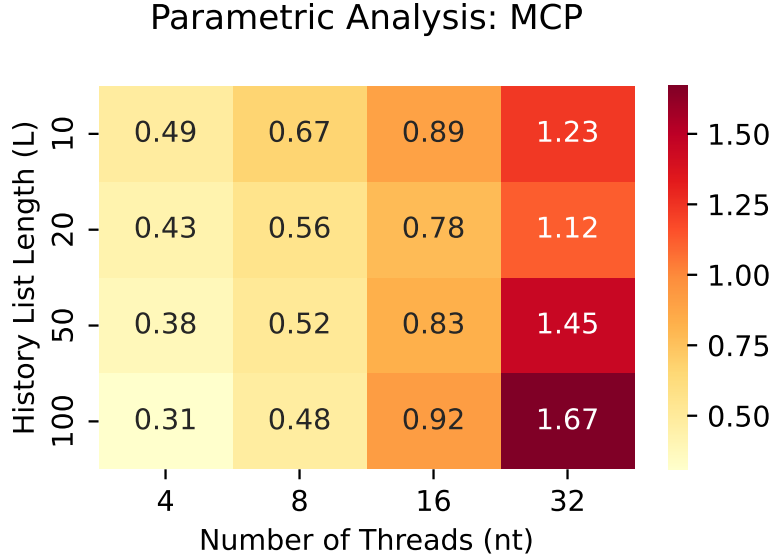


Figure 6. Average gap scores for different history list lengths (L) and number of threads (nt) on MCP

showed different characteristics, with optimal performance achieved at $L = 100$ and $nt = 4$, yielding consistently lower gap scores across all problem sizes. This suggests that MCP benefits from longer history lists but requires fewer parallel threads for effective exploration of the solution space.

Analysis of parameter interactions showed that higher thread counts ($nt \geq 16$) combined with long history lists ($L \geq 100$) often led to degraded performance, especially for larger problem instances. This decline in performance is particularly pronounced in instances of UFLP, where the average gap score increases from 0.29 under optimal parameter configurations to 12.31 when the largest parameter values are employed. This suggests that the exploration-exploitation balance of

the algorithm can be compromised by an excessive degree of parallelism. As a result of these findings, our recommendation is to use moderate thread counts ($4 \leq nt \leq 8$) with problem-specific history list lengths for optimal performance.

5. Conclusion

This study presents a Parallel Late Acceptance Hill-Climbing (PLAHC) algorithm for solving binary-encoded optimization problems (BEOPs), specifically the Uncapacitated Facility Location Problem (UFLP) and the Maximum Cut Problem (MCP). The proposed PLAHC algorithm utilizes parallel processing to improve the performance of the LAHC algorithm.

Our experimental results demonstrate that PLAHC significantly improves upon the sequential LAHC implementation in terms of solution quality and computational efficiency. For UFLP instances, the 8-thread parallel implementation with a history list length (L) of 50 emerged as the best overall configuration, achieving the lowest average gap score of 0.29. For MCP instances, the 4-thread parallel implementation with an L of 100 consistently achieved the best results. Performance comparisons with state-of-the-art algorithms show that PLAHC is highly competitive, often outperforming existing methods.

Despite these results, our extensive testing pointed out several important limitations of the algorithm and its parallel implementation. While significant speedups were achieved, ranging from 3.33x to 10.00x for UFLP and from 2.72x to 9.20x for MCP, the performance improvements exhibit sub-linear scaling as the number of threads increases. This indicates diminishing returns due to communication overhead and algorithmic parts that cannot be perfectly parallelized. Furthermore, for larger problem instances, especially when combined with high degrees of parallelism, very long history lists can lead to suboptimal exploration of the solution space, suggesting a complex interaction between parallelism and search history that requires fine tuning.

The experimental results revealed additional challenges related to thread interference effects and problem instance scaling. Excessive parallelism, especially when combined with longer history lists, can lead to over-exploration or thread interference. This effect was observed for problem instances with extreme configurations, where performance degraded as the number of threads increased. In addition, the scalability of PLAHC algorithm for larger problem sizes remains an open area for further study. For example, while the current study focused on benchmark sets with well-defined sizes and computational budgets, future research could extend these experiments to larger problem instances (e.g., UFLP with more diverse customer and facility sizes, or MCP with larger graph configurations) to better assess the scalability and robustness of the algorithm.

The relationship between the number of threads and performance improvement is non-linear, indicating that further refinement of the parallel implementation is needed. This could include exploring adaptive mechanisms for dynamically adjusting the length of history lists, improving thread coordination, or using alternative parallelization frameworks (e.g., MPI, CUDA) or

hybrid strategies. By extending the experimental setup to a wider range of problem instances and configurations, future research can provide deeper insights into the performance of PLAHC under diverse and challenging conditions, thereby addressing the scalability concerns raised in this study.

Acknowledgments

None.

Funding

This study was supported by the Scientific Research Projects Unit of Karabuk University under grant number KBUBAP-24-ABP-047.

Conflict of interest

The authors declare that they have no conflict of interest regarding the publication of this article.

Author contributions

Conceptualization: All authors

Formal analysis: Emrullah Sonuç

Investigation: All authors

Methodology: Emrullah Sonuç

Supervision: Ender Özcan

Visualization: Emrullah Sonuç

Writing – original draft: Emrullah Sonuç

Writing – review & editing: Ender Özcan

Availability of data

Not applicable.


References

1. Drake JH, Hyde M, Ibrahim K, Ozcan E. A genetic programming hyper-heuristic for the multidimensional knapsack problem. *Kybernetes*. 2014;43(9/10):1500-1511. <https://doi.org/10.1108/K-09-2013-0201>
2. Sonuç E, Özcan E. CUDA-based parallel local search for the set-union knapsack problem. *Knowl-Based Syst*. 2024;112095. <https://doi.org/10.1016/j.knosys.2024.112095>
3. Ahmad A, Alzaidi K, Sari M, Uslu H. Prediction of anemia with a particle swarm optimization-based approach. *Int J Optim Control Theor Appl. (IJOCTA)* 2023;13(2):214-223. <https://doi.org/10.11121/ijocta.2023.1269>
4. Commander CW. Maximum cut problem, MAX-cut. *Encycl Optim*. 2009;2. https://doi.org/10.1007/978-0-387-74759-0_358
5. Glover F, Hanafi S, Guemri O, Crevits I. A simple multi-wave algorithm for the uncapacitated facility location problem. *Front Eng Manag*. 2018;5(4):451-465. <https://doi.org/10.15302/J-FEM-2018038>


6. Talbi EG. *Metaheuristics: From Design to Implementation*. John Wiley & Sons. 2009.
<https://doi.org/10.1002/9780470496916>
7. Al-Betar MA. β -hill climbing: an exploratory local search. *Neural Comput Appl*. 2017;28(Suppl 1):153-168.
<https://doi.org/10.1007/s00521-016-2328-2>
8. Burke EK, Bykov Y. The late acceptance hill-climbing heuristic. *Eur J Oper Res*. 2017;258(1):70-78.
<https://doi.org/10.1016/j.ejor.2016.07.012>
9. Tari S, Basseur M, Goëffon A. Expansion-based Hill-climbing. *Inf Sci*. 2023;649, 119635.
<https://doi.org/10.1016/j.ins.2023.119635>
10. Burke EK, Bykov Y. *The late acceptance hill-climbing heuristic*. University of Stirling, Tech. Rep. 2012.
11. Alparslan S, Sonuç, E.. Solving Static Weapon-Target Assignment Problem using Multi-Start Late Acceptance Hill Climbing. *Curr Trends Comput Sci Appl*. 2024;2(1):23-35.
12. Bazargani M, Lobo FG. Parameter-less late acceptance hill-climbing. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 2017; 219-226.
<https://doi.org/10.1145/3071178.3071225>
13. Goerler A, Schulte F, Voß, S. An application of late acceptance hill-climbing to the traveling purchaser problem. In: *Computational Logistics: 4th International Conference ICCL 2013*, Copenhagen, Denmark, September 25-27, 2013. *Proceedings 4 2013*; 173-183. Springer.
https://doi.org/10.1007/978-3-642-41019-2_13
14. Ghosh M, Kundu T, Ghosh D, Sarkar R. Feature selection for facial emotion recognition using late hill-climbing based memetic algorithm. *Multimed Tools Appl*. 2019;78, 25753-25779.
<https://doi.org/10.1007/s11042-019-07811-x>
15. Turkey A, Sabar NR, Sattar A, Song A. Parallel late acceptance hill-climbing algorithm for the google machine reassignment problem. In: *AI 2016: Advances in Artificial Intelligence: 29th Australasian Joint Conference* Hobart, TAS, Australia, December 5-8, 2016, *Proceedings 29 2016*; 163-174. Springer International Publishing.
https://doi.org/10.1007/978-3-319-50127-7_13
16. Clay S, Mousin L, Veerapen N, Jourdan L. Clahcustom late acceptance hill climbing: First results on tsp. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2021; 1970-1973.
<https://doi.org/10.1145/3449726.3463129>
17. Cao VL, Nicolau M, McDermott J. Late-acceptance and step-counting hill-climbing GP for anomaly detection. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2017; 221-222.
<https://doi.org/10.1145/3067695.3076091>
18. Yuan B, Zhang C, Shao X. A late acceptance hill-climbing algorithm for balancing two-sided assembly lines with multiple constraints. *J Intell Manuf*. 2015;26:159-168.
<https://doi.org/10.1007/s10845-013-0770-x>
19. Fonseca GH, Santos HG, Carrano EG. Late acceptance hill-climbing for high school timetabling. *J Sched*. 2016;19:453-465.
<https://doi.org/10.1007/s10951-015-0458-5>
20. Bolaji AL A, Bamigbola AF, Shola PB. Late acceptance hill climbing algorithm for solving patient admission scheduling problem. *Knowl-Based Syst*. 2018;145:197-206.
<https://doi.org/10.1016/j.knosys.2018.01.017>
21. Erlenkotter D. A dual-based procedure for uncapacitated facility location. *Oper Res*. 1978;26(6):992-1009.
<https://doi.org/10.1287/opre.26.6.992>
22. Efroymsen M, Ray TL. A branch-bound algorithm for plant location. *Oper Res*. 1966;14(3):361-368.
<https://doi.org/10.1287/opre.14.3.361>
23. Akinc U, Khumawala BM. An efficient branch and bound algorithm for the capacitated warehouse location problem. *Manag Sci*. 1977;23(6):585-594.
<https://doi.org/10.1287/mnsc.23.6.585>
24. Schrage L. Implicit representation of variable upper bounds in linear programming. In: *Computational practice in mathematical programming*. 2009; 118-132. Springer.
<https://doi.org/10.1007/BFb0120715>
25. Galvão RD, Raggi LA. A method for solving to optimality uncapacitated location problems. *Ann Oper Res*. 1989;18(1):225-244.
<https://doi.org/10.1007/BF02097805>
26. Hoefer M. Experimental comparison of heuristic and approximation algorithms for uncapacitated facility location. In: *International Workshop on Experimental and Efficient Algorithms*. 2003; 165-178. Springer.
https://doi.org/10.1007/3-540-44867-5_13
27. Monabbati E, Kakhki HT. On a class of subadditive duals for the uncapacitated facility location problem. *Appl Math* 2015;251:118-131.
<https://doi.org/10.1016/j.amc.2014.10.072>
28. Sonuç, E.. Binary crow search algorithm for the uncapacitated facility location problem. *Neural Comput Appl*. 2021;33(21):14669-14685.
<https://doi.org/10.1007/s00521-021-06107-2>
29. Durgut R, Aydin ME. Adaptive binary artificial bee colony algorithm. *Appl Soft Comput*. 2021;101:107054.
<https://doi.org/10.1016/j.asoc.2020.107054>
30. Sonuç, E., Özcan E. An adaptive parallel evolutionary algorithm for solving the uncapacitated facility location problem. *Expert Syst Appl*. 2023;224:119956.
<https://doi.org/10.1016/j.eswa.2023.119956>

31. Aslan M, Pavone M. MBVS: a modified binary vortex search algorithm for solving uncapacitated facility location problem. *Neural Comput Appl.* 2024;36(5):2573-2595.
<https://doi.org/10.1007/s00521-023-09190-9>
32. Gao L, Zeng Y, Dong A. An ant colony algorithm for solving Max-cut problem. *Prog Nat Sci.* 2008;18(9):1173-1178.
<https://doi.org/10.1016/j.pnsc.2008.04.006>
33. Festa P, Pardalos PM, Resende MG, Ribeiro CC. Randomized heuristics for the MAX-CUT problem. *Optim Methods Softw.* 2002;17(6):1033-1058.
<https://doi.org/10.1080/1055678021000090033>
34. Kim YH, Yoon Y, Geem ZW. A comparison study of harmony search and genetic algorithm for the max-cut problem. *Swarm Evol Comput.* 2019;44:130-135.
<https://doi.org/10.1016/j.swevo.2018.01.004>
35. Martí, R., Duarte A, Laguna M. Advanced scatter search for the max-cut problem. *INFORMS J Comput.* 2009;21(1):26-38.
<https://doi.org/10.1287/ijoc.1080.0275>
36. Wu Q, Wang Y, Lü, Z.. A tabu search based hybrid evolutionary algorithm for the max-cut problem. *Appl Soft Comput.* 2015;34:827-837.
<https://doi.org/10.1016/j.asoc.2015.04.033>
37. Kochenberger GA, Hao JK, Lü, Z., Wang H, Glover F. Solving large scale max cut problems via tabu search. *J Heuristics.* 2013;19:565-571.
<https://doi.org/10.1007/s10732-011-9189-8>
38. Akan T, Agahian S, Dehkharghani R. Binbro: Binary battle royale optimizer algorithm. *Expert Syst Appl.* 2022;195:116599.
<https://doi.org/10.1016/j.eswa.2022.116599>
39. Zhu F, Shuai Z, Lu Y et al. oBABC: A one-dimensional binary artificial bee colony algorithm for binary optimization. *Swarm Evol Comput.* 2024;87:101567.
<https://doi.org/10.1016/j.swevo.2024.101567>
40. Šević, I., Jovanovic R, Urošević, D., Davidović, T. Fixed Set Search Applied to the Max-Cut Problem. In: *2024 IEEE 8th Energy Conference (ENERGYCON)*. 2024; 1-6. IEEE.
<https://doi.org/10.1109/ENERGYCON58629.2024.10488777>
41. Beasley JE. OR-Library: distributing test problems by electronic mail. *J Oper Res Soc.* 1990;41(11):1069-1072.
<https://doi.org/10.1057/jors.1990.166>
42. Wiegale A. Biq Mac Library-A collection of Max-Cut and quadratic 0-1 programming instances of medium size. *Preprint.* 2007;51:112-127.

Emrullah Sonuç is currently working as an Associate Professor at Department of Computer Engineering, Karabuk University in Türkiye. He is also a research associate at the Computational Optimization and Learning (COL) Lab in the School of Computer Science at the University of Nottingham, UK. He received M.Sc. and Ph.D. degrees from the Department of Computer Engineering, Karabuk University in 2012 and 2017, respectively. His research interests and activities lie at the interface of computer science, artificial intelligence, and operations research, focusing on metaheuristics, parallel evolutionary algorithms and ensemble learning models.

 <https://orcid.org/0000-0001-7425-6963>

Ender Özcan is a Professor of Computer Science and Operational Research with the Computational Optimization and Learning (COL) Lab in the School of Computer Science at the University of Nottingham, UK. He received his PhD from the Department of Computer and Information Science at Syracuse University, NY, USA in 1998. He worked as a lecturer in the Department of Computer Engineering at Yeditepe University, Istanbul, Turkey from 1998-2007. He established and led the ARTificial Intelligence research group from 2002. He served as the Deputy Head of the Department from 2004-2007. Prof Özcan was appointed as a senior research fellow in 2008 to the EPSRC funded LANCs initiative, which was one of the largest Science and Innovation Rewards given by EPSRC (Engineering and Physical Sciences Research Council, UK). He has been an academic at the University of Nottingham since then. He is currently Director of the Faculty of Science Doctoral Training Centre in Artificial Intelligence.

 <https://orcid.org/0000-0003-0276-1391>

An International Journal of Optimization and Control: Theories & Applications
(<https://accscience.com/journal/ijocta>)



This work is licensed under a Creative Commons Attribution 4.0 International License. The authors retain ownership of the copyright for their article, but they allow anyone to download, reuse, reprint, modify, distribute, and/or copy articles in IJOCTA, so long as the original authors and source are credited. To see the complete license contents, please visit <http://creativecommons.org/licenses/by/4.0/>.