

EvolveSignal: A Large Language Model Powered Coding Agent for Discovering Traffic Signal Control Algorithms

Leizhen Wang, Peibo Duan*, Hao Wang, Yue Wang, Jian Xu, Nan Zheng, Zhenliang Ma*

Abstract—In traffic engineering, the fixed-time traffic signal control remains widely used for its low cost, stability, and interpretability. However, its design depends on hand-crafted formulas (e.g., Webster) and manual re-timing by engineers to adapt to demand changes, which is labor-intensive and often yields suboptimal results under heterogeneous or congested conditions. This paper introduces the EvolveSignal, a large language models (LLMs) powered coding agent to automatically discover new traffic signal control algorithms. We formulate the problem as program synthesis, where candidate algorithms are represented as Python functions with fixed input-output structures, and iteratively optimized through external evaluations (e.g., a traffic simulator) and evolutionary search. Experiments on a signalized intersection demonstrate that the discovered algorithms outperform Webster’s baseline, reducing average delay by 20.1% and average stops by 47.1%. Beyond performance, ablation and incremental analyses reveal that EvolveSignal modifications—such as adjusting cycle length bounds, incorporating right-turn demand, and rescaling green allocations—can offer practically meaningful insights for traffic engineers. This work opens a new research direction by leveraging AI for *algorithm design* in traffic signal control, bridging program synthesis with transportation engineering.

Index Terms—Large Language Models, Traffic Signal Control, Coding Agents, Algorithm Discovery, Evolutionary Search, Heuristic Methods

I. INTRODUCTION

Traffic signal control remains the cost-effective strategy for mitigating urban congestion and improving traffic efficiency. In recent years, artificial intelligence (AI)-based methods have attracted significant attention, especially those utilizing reinforcement learning (RL) [1], [2] and large language models (LLMs) [3], [4]. These data-driven approaches demonstrate the potential to learn high-performance control strategies that can enhance mobility at both the intersection and network levels.

However, most existing studies based on RL and LLMs focus on adaptive traffic signal control (ATSC), which dynamically adjusts signal phases and green times in response to real-time traffic conditions. Although effective, ATSC relies on extensive infrastructure, including high-resolution sensors

(e.g., cameras, radar, inductive loops) and capable signal controllers with strong computational and communication resources. These requirements pose challenges in terms of cost, privacy, and deployment feasibility, limiting the scalability and practical implementation in real-world networks.

In contrast, fixed-time traffic signal control is still widely adopted due to its simplicity, low implementation cost, and operational stability [5], [6]. More importantly, fixed-time control methods are inherently interpretable and transparent, making them easier to adjust and integrate with domain-specific rules. Optimizing fixed-time control plans remains a practically valuable and promising research direction, especially in scenarios where sensing and actuation resources are limited.

Traditional fixed-time control algorithms are typically derived from empirical engineering heuristics, supported by traffic flow theory, queuing models, and optimization methods. In this study, we define a *fixed-time traffic signal control algorithm* as a method that computes a static signal timing plan based on low-resolution or aggregated traffic demand data, without requiring real-time sensing. The output typically includes a fixed cycle length and green time allocation. Among these, Webster’s method [7] is a classic example that uses critical flow ratios and cycle losses to derive closed-form solutions.

While these classical methods are highly interpretable and easy to implement, they are often insufficient to meet the specific operational needs of individual intersections. Due to heterogeneity in traffic demand, geometry, and performance objectives across intersections, a single globally applied strategy (e.g., directly using Webster’s output) may lead to suboptimal outcomes. In practice, traffic engineers are frequently required to fine-tune or adapt these algorithms on a per-intersection basis—adjusting parameters, modifying constraints, or integrating additional domain rules—to better match local conditions and achieve desired performance.

This leads to a critical research question: *Can we move beyond hand-crafted rules and enable automated discovery of fixed-time control algorithms with higher adaptability and performance?*

To address this, we propose leveraging the algorithmic discovery and code generation capabilities of LLMs to optimize fixed-time control algorithms in code (e.g., Python). Specifically, we introduce a novel framework, **EvolveSignal**, which treats a fixed-time signal control algorithm as a function-level program (e.g., a Python function with fixed inputs and outputs) and aims to optimize its internal computation logic—how input variables are processed, intermediate quantities are combined, and outputs are derived. The main contributions of this

Leizhen Wang and Peibo Duan are with the Department of Data Science and AI, Monash University, Melbourne, Australia.

Hao Wang is with the School of Transportation, Southeast University, Nanjing 211189, China.

Yue Wang is with Zhejiang University, Hangzhou 310058, China. He is also with Zhejiang Dahua Technology, Co. Ltd., Hangzhou 310053, China.

Jian Xu is with the Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China.

Nan Zheng is with the Department of Civil and Environmental Engineering, Monash University, Melbourne, Australia.

Zhenliang Ma is with the Department of Civil and Architectural Engineering, KTH Royal Institute of Technology, Stockholm, Sweden.

*Corresponding author: Peibo Duan and Zhenliang Ma.

paper are as follows:

- We formulate fixed-time signal control as a *program synthesis* problem, representing an algorithm as a symbolic function with fixed input–output structure. The focus is on discovering new algorithmic logic, how traffic features are transformed into signal timings.
- We develop **EvolveSignal**, an LLM-powered coding agent that explores, generates, and evaluates candidate programs through simulation-based evaluation and evolutionary search. The framework iteratively refines executable code, enabling interpretable and auditable algorithm discovery.
- We validate the EvolveSignal on synthetic traffic scenarios with heavy traffic conditions, showing significant improvements over Webster’s baseline in terms of delay reduction and stop minimization, while revealing interpretable modifications that can provide useful insights for traffic engineering practice.

The remainder of this paper is organized as follows. Section II reviews related work on conventional and AI-based traffic signal control. Section III introduces the proposed **EvolveSignal** framework. Section IV presents the experimental setup, results, and ablation analyses. Section VI discusses insights and implications. Section VII concludes the paper and outlines future directions.

II. RELATED WORK

A. Conventional traffic signal control methods

Traditional traffic signal control methods for a single intersection can be divided into three groups. The first is fixed-time signal control, such as Webster’s method, which optimizes green time using historical traffic data to minimize delay [7]. The second is actuated signal control, where phases are adjusted based on real-time data from detectors [8]. Both methods, however, struggle to handle dynamic traffic patterns. A third approach, adaptive traffic signal control, dynamically adjusts phases and green times based on real-time conditions [9].

From a control scope perspective, traditional methods can also be classified into single-intersection control, arterial coordination [10], [11], and network-wide control [12]. While these methods have a solid theoretical foundation and are relatively easy to implement, they often require the traffic management engineer to optimize each intersection individually based on local conditions, which is time-consuming, labor-intensive, and costly.

B. AI-based traffic signal control methods

Recently, RL-based approaches have shown significant promise in adaptive traffic signal control, achieving superior performance in terms of reduced delay [2], [13], improved energy efficiency [14], [15], and enhanced safety [16] compared to traditional methods. In parallel, LLM-based methods have shown unique advantages across various transportation domains [17]–[20]. In the context of adaptive traffic signal control, LLMs can deliver even better adaptive control

```
def webster_signal_timing(flow_ratios: List[float], lost_time_per_cycle: float) -> dict:
    """
    Compute fixed-time signal timing using the Webster method.
    """
    Args:
        flow_ratios: List of critical flow ratios for each phase (y_i = q_i / s_i)
        lost_time_per_cycle: Total lost time per cycle (e.g., sum of yellow and all-red times)

    Returns:
        A dictionary containing:
        - 'cycle_length': Optimal cycle length (C)
        - 'green_times': List of green time allocations for each phase

    """
    Y = sum(flow_ratios) # Total flow ratio
    if Y >= 1.0:
        raise ValueError("Total flow ratio too high - fixed-time control not applicable.")

    # Step 1: Compute optimal cycle length (C)
    C = (1.5 * lost_time_per_cycle + 5) / (1 - Y)

    # Step 2: Compute total effective green time (G = C - L)
    effective_green = C - lost_time_per_cycle

    # Step 3: Allocate green time to each phase proportionally
    green_times = [y / Y * effective_green for y in flow_ratios]

    return {
        "cycle_length": round(C, 2),
        "green_times": [round(g, 2) for g in green_times]
    }
```

Fig. 1: The problem definition in Python

performance while providing natural language reasoning and explanations [3], [4]. However, practical deployment remains challenging due to high costs, privacy concerns, and the complex infrastructure required to scale RL and LLM models. This study proposes a novel approach to better harness the potential of AI for traffic signal control, ensuring both scalability and real-world feasibility.

III. METHODOLOGY

A. Problem Definition

This study formulates the optimization of fixed-time traffic signal control algorithms as a program synthesis problem. The objective is to learn an executable function that maps intersection-specific traffic features to a complete signal timing plan. Formally, the problem is defined as:

$$f_{\theta} : X \rightarrow Y, \quad (1)$$

where X denotes the input space, including structural and traffic-related features of an intersection (e.g., traffic volume, geometric layout, and phase configuration); Y represents the output space of fixed-time signal control parameters (e.g., cycle length, green time per phase); and f_{θ} is a parameterized executable program that encodes the logic and structure of the control algorithm using symbolic elements such as variables, operators, formula templates, and conditional statements, as shown in Fig. 1.

Unlike traditional analytical models based on pre-defined rules (e.g., Webster’s formula), this study aims to automatically discover the structure of f_{θ} using a coding agent powered by LLMs and guided by evolutionary search. The LLM generates syntactically valid candidate programs in Python, while a simulation-based evaluation framework iteratively refines them based on traffic performance.

The optimization objective is to maximize the expected score over a distribution of traffic scenarios \mathcal{D} :

$$\max_{f_{\theta}} \mathbb{E}_{x \sim \mathcal{D}} [\text{Score}(f_{\theta}(x))], \quad (2)$$

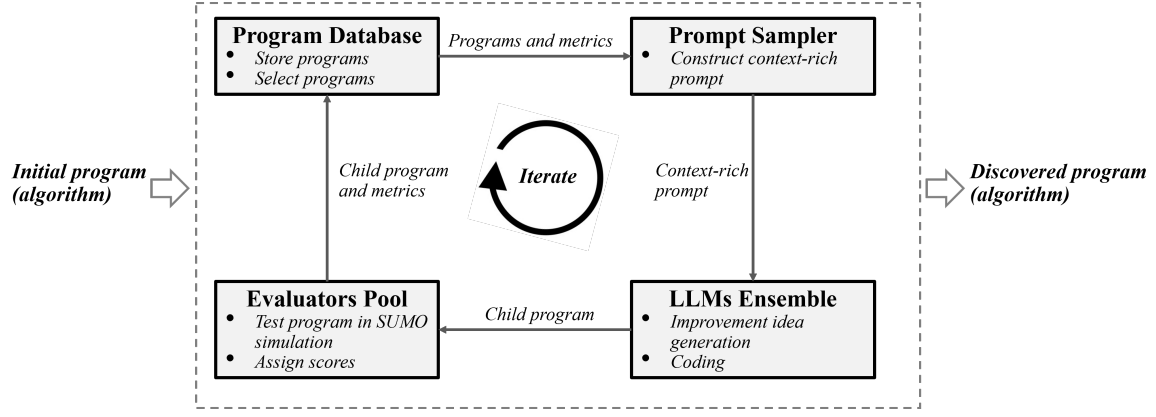


Fig. 2: The EvolveSignal framework

where $\text{Score}(\cdot)$ is a task-specific evaluation metric that quantifies traffic efficiency, such as average throughput, delay reduction, or stop minimization.

This formulation enables the automatic discovery of interpretable, effective, and adaptable fixed-time signal control strategies, suitable for deployment in infrastructure-constrained environments.

B. Overview of the Proposed Framework

Fig. 2 shows the overall framework of **EvolveSignal**, an LLM-powered coding agent designed to evolve fixed-time traffic signal control algorithms. The process begins with an initial program (e.g., an implementation of Webster’s method), which is iteratively modified, simulated, and evaluated until convergence. In each iteration, candidate programs are stored and organized in the *Program Database*, which supports both performance tracking and diversity maintenance. A parent program and several inspiration programs are then passed to the *Prompt Sampler*, which constructs prompts combining code, metrics, and search strategies. These prompts are processed by the *LLMs Ensemble*, generating code-level modifications that act as crossover and mutation operators. The resulting child program is tested in the *Evaluators Pool* using microscopic traffic simulation, and its performance is fed back to the *Program Database*. This loop continues until either the maximum number of iterations is reached or the best-performing algorithm is identified. Detailed descriptions of these modules are provided in Appendix A.

C. Pseudocode for the Proposed Framework

The workflow of EvolveSignal is summarized in Algorithm 1, adapted from the AlphaEvolve framework [21] with implementation modifications from the open-source OpenEvolve project [22]. The pseudocode captures the evolutionary loop of selection, LLM-guided modification, simulation-based evaluation, and population update.

Algorithm 1 Evolutionary Framework for Traffic Fixed-Time Signal Control

- 1: Implement initial program f_{θ_0} (e.g., Webster’s method)
- 2: Initialize Program Database $P \leftarrow \{f_{\theta_0}\}$
- 3: Initialize Evaluators Pool for simulation
- 4: **for** $i = 1$ **to** max_iterations **do**
- 5: Sample parent program $f_{\theta} \in P$
- 6: Select inspiration programs $\mathcal{I} \subset P$ based on top-k and diversity
- 7: Construct prompt P_{prompt} using f_{θ} , \mathcal{I} , and performance metrics
- 8: Sample an LLM model G from LLMs Ensemble
- 9: modifications = $G(P_{\text{prompt}})$
- 10: Apply modifications to generate child program f'_{θ}
- 11: Evaluate f'_{θ} in SUMO [23], compute performance score $\text{Score}(f'_{\theta})$
- 12: Update Program Database with f'_{θ} using MAP-Elites [24]
- 13: **if** termination criteria satisfied **then**
- 14: **break**
- 15: **end if**
- 16: **end for**
- 17: **return** Best discovered program $f_{\theta_{\text{best}}}$

IV. EXPERIMENT

A. Experimental Setup

Experiments were conducted on a simulated four-leg intersection in SUMO (Fig. 3). Each approach is 400 m long with four lanes: one left-turn, two through, and one shared through–right lane. The yellow interval is 3 s and the all-red interval is 1 s. Minimum greens are 20 s for through and 15 s for left turns.

The goal is to evaluate whether the proposed method improves the algorithm’s performance under heavy demand. The intersection operates with four fixed phases: east-west (EW) through, EW left-turn, north-south (NS) through, and NS left-turn phases. Three scenarios (S1–S3) represent distinct demand patterns with overall heavy traffic conditions. Scenario S1 is balanced; S2 emphasizes NS flows; S3 emphasizes EW through and NS left. Demands and Critical Ratio Sum (CRS) values are listed in Table I.

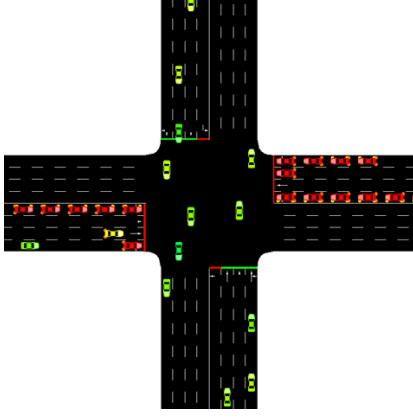


Fig. 3: Simulated isolated intersection in SUMO

TABLE I: Traffic demand scenarios and Critical Ratio Sum (CRS). All values are in *veh/h*, formatted as *through/left/right*.

Scenario	S1	S2	S3
North (N)	1550/210/30	2600/230/30	2500/80/60
South (S)	1450/180/50	2450/250/50	2400/90/70
East (E)	1450/200/40	600/80/40	400/330/150
West (W)	1400/180/40	650/90/40	450/340/120
CRS ^a	0.868	0.865	0.872

^a CRS, based on Webster, reflects demand–capacity ratio; values above 0.85 indicate congestion.

Webster’s Python implementation serves as the initial program. The *LLMs Ensemble* samples from DeepSeek-v3 [25] (40%), DeepSeek-r1 [26] (10%), OpenAI-o4-mini-high (40%), and OpenAI-o3 (10%), with temperature 0.6.

B. Evaluation Metrics

For each scenario, hourly demands were randomly generated in three sets (each 1800 s), and average results were used. Overall performance is the mean across scenarios. Two primary measures were considered: average delay \bar{d} (s/veh) and average stops \bar{s} (stops/veh).

For optimization, they were normalized into scores:

$$S_d = \frac{1}{1 + \bar{d}/100}, \quad S_s = \frac{1}{1 + \bar{s}}, \quad (3)$$

and combined as

$$S_c = 0.5 \times S_d + 0.5 \times S_s. \quad (4)$$

Higher S_c indicates better performance.

V. EXPERIMENT RESULTS

Fig. 4 shows the evolution path of the best discovered program after 300 iterations, along with its performance. The performance generally shows an upward trend with occasional plateaus or even declines during some iterations. Overall, as shown in Table II, the combined score improved from 0.4893 (for the initial program) to 0.5946, which represents a 21.5% improvement. The optimized program outperforms the initial program with a 20.1% reduction in average delay and

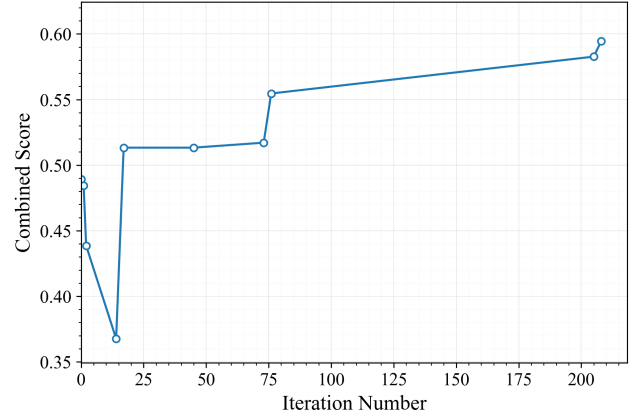


Fig. 4: Evolution path of the best discovered program

TABLE II: Performance Comparison Between the Initial Program and the Best Discovered Program After 300 Iterations

Program	S_d	S_s	S_c
Initial Program	91.79	1.19	0.4893
Discovered Program	73.31	0.63	0.5946
Improvement (%)	-20.1%	-47.1%	+21.5%

a 47.1% reduction in the number of stops, demonstrating the effectiveness of the evolutionary approach.

Fig. 8 and Fig. 9 present the complete Python code of the initial program and the best discovered program. The discovered program introduces several key modifications (highlighted by blue dashed lines in Fig. 9), with the underlying reasoning provided by the LLMs:

- **Cycle Length Bound (CLB).** The maximum cycle length was increased from 130 s to 240 s. As shown in Fig. 10, the LLM argued that a longer cycle allows more effective green time in high-demand scenarios, thereby reducing delay and stops.
- **Right-Turn Inclusion (RTI).** Right-turn flows were added to the through demand calculation. Fig. 11 illustrates how the LLM observed that right-turn vehicles were omitted in the initial program. Given the presence of through–right shared lanes, it is suggested to include right-turn flows to better balance capacity.
- **Shared Lane Factor (SLF).** The contribution of the shared through–right lane was reduced from 0.9 to 0.5. As shown in Fig. 12, the LLM reasoned that a factor of 0.9 overestimated lane capacity, which underestimated the required green time for through movements. Reducing the factor to 0.5 raised their priority, allocating more green to critical through directions.
- **Minimum-Green Feasibility (MGF).** The cycle length was adjusted when the effective green was insufficient to satisfy mandatory minimums. The LLM noted that the initial thresholds (50, 90) were too short to cover the required minimums (20 s per through phase and 15 s per left-turn, plus 16 s for intergreens, totaling 86 s). It accordingly revised the range to (90, 140) to ensure feasibility.

- **Post-Allocation Rescaling (PAR).** Green times were proportionally rescaled to match the effective green budget. As shown in Fig. 11, this avoided both unused “dead” green seconds, which increase delay, and overruns, which reduce capacity for other phases.

a) *Ablation from the discovered program:* Table III shows the ablation results when removing individual modifications from the discovered program. Among all components, **CLB** is the dominant contributor: its removal decreases S_c from 0.5946 to 0.4812 (−19.07%). The benefit of CLB stems from longer cycles diluting the fixed 16s lost time, thereby increasing the share of effective green under heavy demand. Crucially, the improvement does not arise from arbitrarily lengthening cycles but from avoiding performance degradation due to overly restrictive bounds. **RTI** and **SLF** provide secondary yet important gains: removing either reduces S_c by 3.48% and 2.98%, respectively, reflecting their role in balancing green allocation by accounting for right-turn demand and correcting shared-lane capacity. **PAR** offers a modest but stable gain (−0.81% when removed), largely due to rounding effects in high-demand scenarios; it would be more impactful under heterogeneous traffic when some phases approach minimum-green thresholds. Finally, **MGF** has no effect here since all tested cycles are sufficiently long, though it is expected to safeguard feasibility under lighter traffic.

b) *Incremental builds from the initial program:* To further analyze the most effective strategies, Table IV presents an incremental analysis starting from the initial program. Again, **CLB** dominates: adding CLB alone improves S_c by 15.63%. Adding **SLF** yields a minor gain (+1.51%), while adding **RTI** alone surprisingly degrades performance (−2.27%), as incorporating extra demand without cycle or capacity adjustment distorts green allocation. Notably, combining modifications yields synergistic rather than additive effects. For instance, CLB+RTI and CLB+SLF improve S_c by +17.90% and +17.29%, respectively, exceeding the sum of individual contributions. This suggests a coupling between demand modeling and cycle flexibility: RTI or SLF only becomes effective when longer cycles allow sufficient redistribution of green. These findings highlight two insights: (i) some modifications are detrimental in isolation, and (ii) evolutionary search is crucial for identifying synergistic combinations that outperform individual strategies. Beyond performance, this also illustrates how a coding-agent framework can uncover non-intuitive design patterns that may be difficult for human engineers to identify through manual, additive reasoning.

TABLE III: Ablation study of the optimized program by removing individual modifications. (“w/o” denotes removal of a component).

Variant	\bar{d}	\bar{s}	S_c	Change
Optimized Program	73.31	0.63	0.5946	0.00%
w/o CLB	89.32	1.30	0.4812	−19.07%
w/o RTI	81.00	0.68	0.5739	−3.48%
w/o SLF	76.46	0.70	0.5769	−2.98%
w/o MGF	73.31	0.63	0.5946	0.00%
w/o PAR	73.64	0.66	0.5898	−0.81%

TABLE IV: Incremental analysis by progressively adding modifications to the initial program. (“+” denotes addition of a component)

Variant	\bar{d}	\bar{s}	S_c	Change
Initial Program	91.79	1.19	0.4893	0.00%
+ CLB	84.01	0.70	0.5658	+15.63%
+ RTI	90.81	1.31	0.4782	−2.27%
+ SLF	87.27	1.18	0.4967	+1.51%
+ CLB + RTI	76.46	0.70	0.5769	+17.90%
+ RTI + SLF	87.56	1.20	0.4935	+0.86%
+ CLB + SLF	81.00	0.68	0.5739	+17.29%

VI. DISCUSSION

A. Types of LLM-Generated Modifications

Across iterations, most LLM-generated modifications fall into three categories: *heuristic rules*, *traffic-engineering principles*, and *Hyper-parameter adjustments*. The first introduces conditional rules (e.g., `if--else` branches) to adapt logic under different demand conditions. The second leverages domain knowledge, such as refining critical movement analysis or adjusting green allocation to reflect imbalances. The third modifies parameters such as cycle length, saturation flow, or shared-lane factors; unlike conventional tuning, these changes are typically justified with reasoning about expected effects. Importantly, not all useful modifications remain in the best discovered program. As ablation and incremental analyses show, some are only beneficial in combination, while others degrade performance in isolation. Even discarded strategies (e.g., Fig. 14) can be valuable, as refinements in saturation flow or demand-scaled allocation highlight potential directions for practical design.

B. Exploration and Diversity

Increasing iterations and adding more models to the *LLMs Ensemble* can further improve outcomes. The extended run in Fig. 15 achieved slightly higher scores, reflecting the benefit of broader exploration. Different LLMs often suggest distinct modifications, enhancing diversity and the chance of synergy. However, more modifications also make code harder to interpret and maintain, which may hinder deployment. To balance this, we focused on a relatively interpretable discovered program, while recognizing that multiple runs may produce different yet comparable solutions. Under a fixed prompt template, the same LLM (e.g., DeepSeek-r1) tends to generate similar families of modifications even with relative large temperatures, revealing stable biases; this consistency implies that greater diversity must come from varying prompts or combining multiple LLMs.

C. Interpretability and Practical Value

Although the internal reasoning of LLMs is black-box, the generated modifications are accompanied by natural language explanations that align with traffic engineering logic. This provides interpretability absent in conventional black-box optimization. Moreover, since outputs are executable Python code, practitioners can directly inspect and adapt modifications. This dual transparency—conceptual reasoning plus explicit

code—offers practical value, enabling engineers to retain or refine the most useful strategies. In practice, such human-in-the-loop supervision may prove as valuable as raw performance gains, bridging AI-driven synthesis with domain expertise in signal control.

VII. CONCLUSION

This paper presented **EvolveSignal**, an LLM-powered framework for the automated discovery of fixed-time traffic signal control algorithms. By formulating fixed-time optimization as a program synthesis problem, the framework moves beyond parameter tuning and enables the generation of new algorithmic logics expressed as executable code. Experiments on a congested intersection showed that the discovered program outperformed Webster’s method, reducing average delay by 20.1% and stops by 47.1%.

Ablation and incremental analyses revealed that not all modifications are effective in isolation, but certain combinations yield synergistic improvements. Even discarded modifications during evolution provided interpretable design insights, showing how LLM-driven exploration can inspire new traffic engineering heuristics. Furthermore, the framework enhances interpretability by producing human-readable code together with natural language reasoning, allowing practitioners to selectively refine and adapt the results.

The current study is limited to a single isolated intersection with fixed-time control. Future work will extend EvolveSignal to larger-scale networks and design algorithms tailored to more complex intersection geometries. Incorporating additional LLMs and human-in-the-loop strategies will further enhance diversity and practical utility. Overall, this study positions LLM-powered program synthesis as a promising and interpretable pathway for advancing traffic signal control in infrastructure-constrained environments.

ACKNOWLEDGMENTS

The work was supported by start-up funds with No. MSRI8001004 and No. MSRI9002005 at Monash University and TRENOP center and Digital Futures at KTH Royal Institute of Technology, Sweden.

REFERENCES

- [1] H. Wei, G. Zheng, V. Gayah, and Z. Li, “A survey on traffic signal control methods,” *arXiv preprint arXiv:1904.08117*, 2019.
- [2] L. Wang, Z. Ma, C. Dong, and H. Wang, “Human-centric multimodal deep (hmd) traffic signal control,” *IET Intelligent Transport Systems*, vol. 17, no. 4, pp. 744–753, 2023.
- [3] M. Movahedi and J. Choi, “The crossroads of llm and traffic control: A study on large language models in adaptive traffic signal control,” *IEEE Transactions on Intelligent Transportation Systems*, 2024.
- [4] S. Lai, Z. Xu, W. Zhang, H. Liu, and H. Xiong, “Llmight: Large language models as traffic signal control agents,” in *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*, 2025, pp. 2335–2346.
- [5] W. Zhou, Y. Wang, M. Liu, T. Liu, P. Zhang, and Z. Ma, “Traffic signal phase and timing estimation using trajectory data from radar vision integrated camera,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 25, no. 11, pp. 18 279–18 291, Nov 2024.
- [6] Y. Wang, M. Zhou, G. Huang, R. Zhuo, C. Yi, and Z. Ma, “Chat2spat: A large language model based tool for automating traffic signal control plan management,” *arXiv preprint arXiv:2507.05283*, 2025.
- [7] F. V. Webster, “Traffic signal settings,” Tech. Rep., 1958.
- [8] M. Fellendorf, “Vissim: A microscopic simulation tool to evaluate actuated signal control including bus priority,” in *64th Institute of transportation engineers annual meeting*, vol. 32. Springer Berlin/Heidelberg, Germany, 1994, pp. 1–9.
- [9] Y. Wang, X. Yang, H. Liang, and Y. Liu, “A review of the self-adaptive traffic signal control system based on future traffic environment,” *Journal of Advanced Transportation*, vol. 2018, no. 1, p. 1096123, 2018.
- [10] N. H. Gartner, S. F. Assman, F. Lasaga, and D. L. Hou, “A multi-band approach to arterial traffic signal optimization,” *Transportation Research Part B: Methodological*, vol. 25, no. 1, pp. 55–74, 1991.
- [11] H. Wang and X. Peng, “Coordinated control model for oversaturated arterial intersections,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 12, pp. 24 157–24 175, 2022.
- [12] P. Hunt, D. Robertson, R. Bretherton, and M. C. Royle, “The scout on-line traffic signal optimisation technique,” *Traffic Engineering & Control*, vol. 23, no. 4, 1982.
- [13] T. Chu, J. Wang, L. Codecà, and Z. Li, “Multi-agent deep reinforcement learning for large-scale traffic signal control,” *IEEE transactions on intelligent transportation systems*, vol. 21, no. 3, pp. 1086–1095, 2019.
- [14] L. Koch, T. Brinkmann, M. Wegener, K. Badalian, and J. Andert, “Adaptive traffic light control with deep reinforcement learning: An evaluation of traffic flow and energy consumption,” *IEEE transactions on intelligent transportation systems*, vol. 24, no. 12, pp. 15 066–15 076, 2023.
- [15] X. Peng, S. Chen, H. Gao, H. Wang, and H. M. Zhang, “Combat urban congestion via collaboration: Heterogeneous gnn-based marl for coordinated platooning and traffic signal control,” *IEEE Transactions on Intelligent Transportation Systems*, 2025.
- [16] Y. Gong, M. Abdel-Aty, J. Yuan, and Q. Cai, “Multi-objective reinforcement learning approach for improving safety at intersections with adaptive traffic signal control,” *Accident Analysis & Prevention*, vol. 144, p. 105655, 2020.
- [17] Z. Ma, L. Wang, Z. Qin, and Y. Ling, “Large language models for urban transportation: Basics, methods, and applications,” in *Mobility Patterns, Big Data and Transportation Analytics*, 2nd ed. Elsevier, 2025.
- [18] L. Wang, P. Duan, Z. He, C. Lyu, X. Chen, N. Zheng, L. Yao, and Z. Ma, “Ai-driven day-to-day route choice,” *arXiv preprint arXiv:2412.03338*, 2024.
- [19] Z. Qin, L. Wang, F. C. Pereira, and Z. Ma, “A foundational individual mobility prediction model based on open-source large language models,” *arXiv preprint arXiv:2503.16553*, 2025.
- [20] Z. Qin, P. Zhang, L. Wang, and Z. Ma, “Lingotrip: Spatiotemporal context prompt driven large language model for individual trip prediction,” *Journal of Public Transportation*, vol. 27, p. 100117, 2025.
- [21] A. Novikov, N. Vü, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. Ruiz, A. Mehrabian *et al.*, “Alphaevolve: A coding agent for scientific and algorithmic discovery,” *arXiv preprint arXiv:2506.13131*, 2025.
- [22] A. Sharma, “Openevolve: an open-source evolutionary coding agent,” 2025. [Online]. Available: <https://github.com/codelion/openevolve>
- [23] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner, “Microscopic traffic simulation using sumo,” in *2018 21st international conference on intelligent transportation systems (ITSC)*. Ieee, 2018, pp. 2575–2582.
- [24] J.-B. Mouret and J. Clune, “Illuminating search spaces by mapping elites,” *arXiv preprint arXiv:1504.04909*, 2015.
- [25] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, “Deepseek-v3 technical report,” *arXiv preprint arXiv:2412.19437*, 2024.
- [26] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, “Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning,” *arXiv preprint arXiv:2501.12948*, 2025.

APPENDIX

A. Detailed Description of Framework Modules

This appendix provides extended details of the four modules in the proposed EvolveSignal framework (Section III): *Program Database*, *Prompt Sampler*, *LLMs Ensemble*, and *Evaluators Pool*. These were omitted from the main text due to space constraints but are included here for completeness.

Program Database. The *Program Database* maintains the population of candidate programs during evolution. For each program, it stores performance metrics, evolutionary history, and metadata. To balance exploration and exploitation, the population is organized using the MAP-Elites [24] algorithm, which preserves diversity while promoting high-performing solutions. In each iteration, one parent program and several inspiration programs are sampled for modification.

Prompt Sampler. The *Prompt Sampler* constructs the prompts provided to the LLMs. It generates a system message (see Fig. 5) that defines the LLM’s role and the task it is expected to perform, along with a user message (Fig. 6) that consolidates the parent program, inspiration programs, and their performance metrics. To support different search strategies, a flexible template mechanism allows prompts to specify either local code modifications (analogous to differential evolution) or full rewrites. This ensures that the LLMs receive both structural and performance context, guiding them to produce relevant modifications.

LLMs Ensemble. The *LLMs Ensemble* integrates multiple LLMs (e.g., DeepSeek-v3 [25], DeepSeek-r1 [26]). In each iteration, one model is sampled to generate suggestions. This diversity increases the likelihood of discovering effective modifications, as different models emphasize different reasoning paths. Fig. 7 shows an example response. Unlike conventional crossover and mutation restricted to parameter values, LLMs can propose structural code-level changes, expanding the search space and enabling novel solutions.

Evaluators Pool. The *Evaluators Pool* assesses the quality of generated programs using microscopic traffic simulation in SUMO [23]. Each program generates a fixed-time signal control plan (e.g., green allocations per phase) based on the given scenario inputs. This strategy is then executed in simulation, and performance metrics such as average delay and number of stops are recorded. The evaluation results are returned to the *Program Database*, where they influence program retention and guide subsequent modifications.

B. Prompt and Response Examples

This section provides supplementary examples of prompts and LLM responses used in EvolveSignal. Figs. 5 and 6 show representative system and user message templates, while Fig. 7 illustrates a typical LLM response. Figs. 8 and 9 present the initial and discovered programs in Python, with key modifications highlighted. Figs. 10–13 provide detailed examples of individual modifications generated during evolution. Finally, Figs. 14 and 15 illustrate additional results discussed in Section VI.

[System message]

You are a traffic signal control expert and software engineer. Your role is to design and improve signal timing algorithms for fixed-time, four-phase intersections to minimize vehicle delay and stops. The four signal phases are:

NS_through: north-south through
 NS_left: north-south left turn
 EW_through: east-west through
 EW_left: east-west left turn

Intersection setup:

Each approach has 4 lanes: 1 dedicated left-turn lane, 2 through lanes, and 1 shared through-right lane
 Yellow time: 3 seconds (fixed)
 All-red time: 1 second (fixed)
 Minimum green time: 20 seconds for through movements, 15 seconds for left-turn movements

Evaluation Scenarios:

Your algorithm will be evaluated across three simulation scenarios: Scenario 1: Moderate and balanced traffic demand across all directions. Scenario 2: Heavy demand on NS through and NS left-turn movements. Scenario 3: Heavy demand on EW through and NS left-turn movements.

Optimization Goal:

Your algorithm (Python function) will be evaluated using three metrics: delay score, stops score, and combined score - all of which are higher when performance is better. The metrics are computed as follows (where average_delay is in seconds and average_stops is the average number of stops per vehicle):

$$\text{delay_score} = 1.0 / (1.0 + \text{average_delay} / 100.0)$$

$$\text{stops_score} = 1.0 / (1.0 + \text{average_stops} / 1.0)$$

$$\text{combined_score} = 0.5 * \text{delay_score} + 0.5 * \text{stops_score}$$

Fig. 5: Example system message prompt template

[User message]

Current Program Information

- Current performance metrics: {metrics}
- Areas identified for improvement: {improvement_areas}

{artifacts}

Program Evolution History

{evolution_history}

Current Program

```
```{language}
{current_program}
```
```

Task

You may apply any reasonable strategy to optimize the algorithm, including:

- Traffic engineering principles (e.g., flow ratio balancing, critical movement analysis, checking whether timing constraints affect the intended signal plan, and addressing them with or without iteration)
- Rule-based or logic-driven heuristics (e.g., condition-based phase time adjustments, logic-driven overrides)
- Parameter tuning (e.g., cycle length, saturation flow, other algorithmic hyperparameters)
- Ideas inspired by other domains (e.g., task scheduling, resource allocation, control theory)
- Lightweight refinements (e.g., bottleneck-priority tuning, small rule-based refinements that require low implementation effort)

Randomly select one strategy category from the list above. Then, under that strategy, propose one or several specific changes that are most likely to improve the algorithm's performance.

You MUST use the exact SEARCH/REPLACE diff format shown below to indicate changes:

```
<<<<<<< SEARCH
# Original code to find and replace (must match exactly)
=====
# New replacement code
>>>>>>> REPLACE
```

Example of valid diff format:

```
<<<<<<< SEARCH
for i in range(m):
    for j in range(p):
        for k in range(n):
            C[i, j] += A[i, k] * B[k, j]
=====
# Reorder loops for better memory access pattern
for i in range(m):
    for k in range(n):
        for j in range(p):
            C[i, j] += A[i, k] * B[k, j]
>>>>>>> REPLACE
```

You can suggest multiple changes, they should be mutually supportive or at least non-conflicting. Each SEARCH section must exactly match code in the current program. Be thoughtful about your changes and explain your reasoning thoroughly.

IMPORTANT: Do not rewrite the entire program - focus on targeted improvements.

Fig. 6: Example user message prompt template

[LLM Response]

I'll focus on "Traffic engineering principles" to optimize the algorithm. The key improvements will be around better handling of cycle length calculation and green time allocation to balance demand across phases.

[Improvement idea 1]

1. First, let's improve the cycle length calculation to better handle edge cases and ensure smoother transitions:

```
<<<<<<< SEARCH
    min_cycle_length = total_min_green + lost_time
    if total_critical_ratio < 1:
        C0 = (1.5 * lost_time + 5) / (1 - total_critical_ratio)
    else:
        C0 = cycle_length_max # in case of saturation, set to max cycle length (but will be constrained by
min_cycle_length and max)

    # Constrain the cycle length by min, max, and the minimum required by the min green times
    cycle_length = max(cycle_length_min, min(cycle_length_max, round(C0), min_cycle_length))
=====
    min_cycle_length = total_min_green + lost_time
    if total_critical_ratio < 0.9: # More conservative threshold for stability
        C0 = (1.5 * lost_time + 5) / (1 - min(total_critical_ratio, 0.95)) # Cap ratio at 0.95 to prevent extreme cycle lengths
    else:
        # For high demand, use Webster's formula with upper bound
        C0 = min(cycle_length_max, max(min_cycle_length, 4 * lost_time / (1 - 0.95)))

    # Constrain the cycle length with smoother transitions
    cycle_length = max(cycle_length_min, min(cycle_length_max, round(C0), min_cycle_length))
>>>>>>> REPLACE
```

[Improvement idea 2]

2. Next, let's improve the green time allocation to better handle unbalanced demands:

```
<<<<<<< SEARCH
(current_code_snippet)
=====
(modified_code_snippet)
>>>>>>> REPLACE
```

[Summary]

These changes improve the algorithm by:

1. Making cycle length calculation more stable by capping critical flow ratio at 0.95 and using Webster's formula for high demand cases
2. Adding movement-type awareness to green time allocation (through movements get higher priority)
3. Implementing demand-sensitive priority weights to better handle unbalanced scenarios
4. Adding smoother transitions between different demand conditions

The improvements should particularly help in Scenarios 2 and 3 where demand is unbalanced, while maintaining good performance in Scenario 1. The priority weighting system helps ensure that high-demand movements get adequate green time while still respecting minimum green constraints.

The cycle length calculation changes prevent extreme cycle lengths that could lead to excessive delays while still being responsive to demand. The 0.9 threshold for using Webster's formula provides a buffer zone for more stable operation.

Fig. 7: Representative LLM response (DeepSeek-v3)

[initial_program.py (Part 1)]

```

"""Traffic-signal control (fixed lane layout and phase sequence)"""

# === Fixed lane configuration ===
LANES_THROUGH = 2      # Number of dedicated through lanes per approach
LANES_LEFT = 1          # Number of dedicated left-turn lanes per approach
LANES_THROUGH_RIGHT = 1 # Number of shared through-right lanes per approach

# === Fixed control parameters ===
MIN_GREEN_THROUGH = 20 # Minimum green time for through phases (s)
MIN_GREEN_LEFT = 15    # Minimum green time for left-turn phases (s)
YELLOW_TIME = 3        # Yellow time per phase (s)
ALL_RED_TIME = 1       # All-red time per phase (s)

# EVOLVE-BLOCK-START
import numpy as np

def signal_timing_algorithm(traffic_flows):
    """
    Compute fixed-time signal timing.
    Fixed phase sequence: [NS_through, NS_left, EW_through, EW_left]
    Lane layout is fixed as defined above.

    Args:
        traffic_flows (dict): Movement-specific demand in veh/h, e.g.
            {'N_through': 300, 'N_left': 150, 'N_right': 100, ...}

    Returns:
        dict: Timing plan (green_times) and metadata.
    """
    cycle_length_min, cycle_length_max = 60, 130 # Minimum and maximum cycle length
    sat_flow_through, sat_flow_left = 1800, 1400 # Saturation flow for through lanes (veh/h/lane)

    # 1. Compute lane-level flow ratios for each movement
    movement_flow_ratio = {}
    for d in ['N', 'S', 'E', 'W']:
        q_th = traffic_flows.get(f'{d}_through', 0)
        q_lf = traffic_flows.get(f'{d}_left', 0)

        effective_th_lanes = LANES_THROUGH + 0.9 * LANES_THROUGH_RIGHT
        movement_flow_ratio[f'{d}_through'] = q_th / (effective_th_lanes * sat_flow_through)
        movement_flow_ratio[f'{d}_left'] = q_lf / (LANES_LEFT * sat_flow_left)

    # 2. Determine the critical flow ratio for each phase
    critical_flow_ratio = {
        'NS_through': max(movement_flow_ratio['N_through'],
            movement_flow_ratio['S_through']),
        'NS_left': max(movement_flow_ratio['N_left'],
            movement_flow_ratio['S_left']),
        'EW_through': max(movement_flow_ratio['E_through'],
            movement_flow_ratio['W_through']),
        'EW_left': max(movement_flow_ratio['E_left'],
            movement_flow_ratio['W_left']),
    }

```

[initial_program.py (Part 2)]

```

    # 3. Compute cycle length and total effective green time
    lost_time = 4 * (YELLOW_TIME + ALL_RED_TIME)
    total_critical_ratio = sum(critical_flow_ratio.values())
    C0 = (1.5 * lost_time + 5) / (1 - total_critical_ratio) if total_critical_ratio < 1 else
    cycle_length_max
    cycle_length = max(cycle_length_min, min(cycle_length_max, round(C0)))
    effective_green_time = cycle_length - lost_time

    # 4. Allocate green time to each phase (with minimum green constraints)
    theoretical_green = {}
    actual_green_time = {}
    for phase, y in critical_flow_ratio.items():
        min_g = MIN_GREEN_THROUGH if 'through' in phase else MIN_GREEN_LEFT
        if total_critical_ratio == 0:
            theoretical_green[phase] = min_g
            actual_green_time[phase] = min_g
        else:
            g = round(y / total_critical_ratio * effective_green_time)
            theoretical_green[phase] = g
            actual_green_time[phase] = max(min_g, g)

    # 5. Calculate actual green time proportions per phase
    total_actual_green = sum(actual_green_time.values())
    green_time_ratio = {
        phase: round(g / total_actual_green, 4)
        for phase, g in actual_green_time.items()
    }

    return {
        'green_times': actual_green_time,
        'metadata': {
            'lost_time': lost_time,
            'critical_flow_ratio': critical_flow_ratio,
            'green_time_ratio': green_time_ratio,
            'theoretical_cycle_length': round(C0, 4),
            'actual_cycle_length': cycle_length,
            'theoretical_green': theoretical_green,
            'actual_green': actual_green_time,
        }
    }
# EVOLVE-BLOCK-END.

```

Fig. 8: Implementation of the initial program (initial_program.py)

[optimized_program.py (Part 1)]

```

"""Traffic-signal control (fixed lane layout and phase sequence)"""

# === Fixed lane configuration ===
LANES_THROUGH = 2      # Number of dedicated through lanes per approach
LANES_LEFT = 1         # Number of dedicated left-turn lanes per approach
LANES_THROUGH_RIGHT = 1 # Number of shared through-right lanes per approach

# === Fixed control parameters ===
MIN_GREEN_THROUGH = 20 # Minimum green time for through phases (s)
MIN_GREEN_LEFT = 15    # Minimum green time for left-turn phases (s)
YELLOW_TIME = 3        # Yellow time per phase (s)
ALL_RED_TIME = 1        # All-red time per phase (s)

# EVOLVE-BLOCK-START
import numpy as np

def signal_timing_algorithm(traffic_flows):
    """
    Compute fixed-time signal timing.
    Fixed phase sequence: [NS_through, NS_left, EW_through, EW_left]
    Lane layout is fixed as defined above.

    Args:
        traffic_flows (dict): Movement-specific demand in veh/h, e.g.
            {'N_through': 300, 'N_left': 150, 'N_right': 100, ...}

    Returns:
        dict: Timing plan (green_times) and metadata.
    """
    # Increase cycle length bounds to satisfy minimum greens and offer flexibility
    cycle_length_min, cycle_length_max = 60, 240 # Expanded cycle length range for
    heavy demand scenarios to reduce stops
    sat_flow_through, sat_flow_left = 1800, 1400 # Adjusted saturation flows to more
    conservative values for improved green allocation under heavy demands

    # 1. Compute lane-level flow ratios for each movement
    movement_flow_ratio = {}
    for d in ['N', 'S', 'E', 'W']:
        # include right-turn demand as part of through capacity usage
        q_th = traffic_flows.get(f'{d}_through', 0) + traffic_flows.get(f'{d}_right', 0)
        q_lf = traffic_flows.get(f'{d}_left', 0)

        # Treat the shared through-right lane as providing only half of a true through lane
        effective_th_lanes = LANES_THROUGH + 0.5 * LANES_THROUGH_RIGHT
        movement_flow_ratio[f'{d}_through'] = q_th / (effective_th_lanes *
        sat_flow_through)
        movement_flow_ratio[f'{d}_left'] = q_lf / (LANES_LEFT * sat_flow_left)

    # 2. Determine the critical flow ratio for each phase
    critical_flow_ratio = {
        'NS_through': max(movement_flow_ratio['N_through'],
        movement_flow_ratio['S_through']),
        'NS_left': max(movement_flow_ratio['N_left'], movement_flow_ratio['S_left']),
        'EW_through': max(movement_flow_ratio['E_through'],
        movement_flow_ratio['W_through']),
        'EW_left': max(movement_flow_ratio['E_left'], movement_flow_ratio['W_left']),
    }

```

[optimized_program.py (Part 2)]

```

# 3. Compute cycle length and total effective green time
lost_time = 4 * (YELLOW_TIME + ALL_RED_TIME)
total_critical_ratio = sum(critical_flow_ratio.values())
C0 = (1.5 * lost_time + 5) / (1 - total_critical_ratio) if total_critical_ratio < 1 else
cycle_length_max
cycle_length = max(cycle_length_min, min(cycle_length_max, round(C0)))
effective_green_time = cycle_length - lost_time
# Ensure the cycle length can accommodate all mandatory minimum greens
min_green_total = 2 * MIN_GREEN_THROUGH + 2 * MIN_GREEN_LEFT
if effective_green_time < min_green_total:
    cycle_length = min(cycle_length_max, min_green_total + lost_time)
    effective_green_time = cycle_length - lost_time # 4. Allocate green time to each phase
(with minimum green constraints)
theoretical_green = {}
actual_green_time = {}
for phase, y in critical_flow_ratio.items():
    min_g = MIN_GREEN_THROUGH if 'through' in phase else MIN_GREEN_LEFT
    if total_critical_ratio == 0:
        theoretical_green[phase] = min_g
        actual_green_time[phase] = min_g
    else:
        g = round(y / total_critical_ratio * effective_green_time)
        theoretical_green[phase] = g
        actual_green_time[phase] = max(min_g, g)

# 5. Adjust actual green times to exactly fill the effective green budget
total_actual_green = sum(actual_green_time.values())
if total_actual_green and total_actual_green != effective_green_time:
    scale = effective_green_time / total_actual_green
    print('scale=', scale)
    for phase in actual_green_time:
        min_g = MIN_GREEN_THROUGH if 'through' in phase else MIN_GREEN_LEFT
        actual_green_time[phase] = max(min_g, round(actual_green_time[phase] * scale))

# 6. Calculate actual green time proportions per phase
total_actual_green = sum(actual_green_time.values())
green_time_ratio = {
    phase: round(g / total_actual_green, 4)
    for phase, g in actual_green_time.items()
}

return {
    'green_times': actual_green_time,
    'metadata': {
        'lost_time': lost_time,
        'critical_flow_ratio': critical_flow_ratio,
        'green_time_ratio': green_time_ratio,
        'theoretical_cycle_length': round(C0, 4),
        'actual_cycle_length': cycle_length,
        'theoretical_green': theoretical_green,
        'actual_green': actual_green_time,
    }
}

# EVOLVE-BLOCK-END

```

Fig. 9: Implementation of the discovered program (discovered_program.py), with key modifications highlighted (blue dashed lines)

[LLM Response]

Strategy category: **Hyperparameter tuning**.

By lowering the assumed saturation flows and giving the algorithm more headroom on maximum cycle length, the model will allocate more green time under high-demand scenarios—reducing stops and delays.

- 1) Increase `cycle_length_max` to allow longer cycles when demands are heavy.
- 2) Reduce `sat_flow_through` and `sat_flow_left` to produce higher critical flow ratios (and thus longer, more appropriate green allocations) under the same raw volumes.

<<<<<<< SEARCH

cycle_length_min, cycle_length_max = 60, 200 # Tuned cycle length range for reduced delay under lighter loads and extra headroom under peaks

=====

cycle_length_min, cycle_length_max = 60, 240 # Expanded cycle length range for heavy demand scenarios to reduce stops

>>>>>>> REPLACE

<<<<<<< SEARCH

sat_flow_through, sat_flow_left = 1900, 1500 # Updated saturation flows to better reflect true lane capacities and permit more green time when needed

=====

sat_flow_through, sat_flow_left = 1800, 1400 # Adjusted saturation flows to more conservative values for improved green allocation under heavy demands

>>>>>>> REPLACE

Fig. 10: Example LLM response introducing the CLB modification

[LLM Response]

I've chosen a **rule-based (logic-driven) heuristic strategy** to ensure we're (1) counting all turning movements in our flow ratios and (2) making full use of the available green time. Both changes together will better balance capacity and reduce wasted time.

```
<<<<<<< SEARCH
    q_th = traffic_flows.get(f'{d}_through', 0)
=====
    # include right-turn demand as part of through capacity usage
    q_th = traffic_flows.get(f'{d}_through', 0) + traffic_flows.get(f'{d}_right', 0)
>>>>>>> REPLACE
```

This ensures right-turn volumes aren't ignored—otherwise that demand goes unserved, increasing delay and stops on shared lanes.

```
<<<<<<< SEARCH
    # 5. Calculate actual green time proportions per phase
    total_actual_green = sum(actual_green_time.values())
    green_time_ratio = {
        phase: round(g / total_actual_green, 4)
        for phase, g in actual_green_time.items()
    }
=====
    # 5. Adjust actual green times to exactly fill the effective green budget
    total_actual_green = sum(actual_green_time.values())
    if total_actual_green and total_actual_green != effective_green_time:
        scale = effective_green_time / total_actual_green
        for phase in actual_green_time:
            min_g = MIN_GREEN_THROUGH if 'through' in phase else MIN_GREEN_LEFT
            actual_green_time[phase] = max(min_g, round(actual_green_time[phase] * scale))

    # 6. Calculate actual green time proportions per phase
    total_actual_green = sum(actual_green_time.values())
    green_time_ratio = {
        phase: round(g / total_actual_green, 4)
        for phase, g in actual_green_time.items()
    }
>>>>>>> REPLACE
```

By scaling the post-minimum allocations to exactly match the cycle's effective green time, we avoid both “dead” unused green seconds (which inflate delay) and inadvertent overruns (which dilute capacity for other movements).

Fig. 11: Example LLM response introducing the RTI and PAR modifications

[LLM Response]

Selected strategy category (random draw): **Hyper-parameter tuning**

Treat the shared through-right lane as providing only 50 % of a regular through lane (instead of 90 %).

- Rationale: Under heavy traffic, right-turns usually discharge faster than through vehicles. Counting the shared lane as almost a full through lane (0.9) over-estimates the effective capacity, causing the algorithm to under-allocate green to the critical through phases. Lowering the factor to 0.5 increases the calculated flow ratio for those movements, so extra green is given where it is most needed.

Diff showing the exact code edits:

```
<<<<<<< SEARCH
    effective_th_lanes = LANES_THROUGH + 0.9 * LANES_THROUGH_RIGHT
=====
    # Treat the shared through-right lane as providing only half of a true through lane
    effective_th_lanes = LANES_THROUGH + 0.5 * LANES_THROUGH_RIGHT
>>>>>>> REPLACE
```

Fig. 12: Example LLM response introducing the SLF modification

[LLM Response]

Chosen strategy category (randomly selected): **Traffic-engineering principles** – verification of timing constraints and ensuring cycle length feasibility.

Problem observed:

The algorithm can select a cycle that is too short to hold the mandatory minimum greens (20 s for each through + 15 s for each left = 70 s). With the fixed lost time of 16 s, the cycle must be at least 86 s.

When the computed cycle is shorter (e.g., 50 s), the later scaling step fails to reconcile the mismatch and produces an internally inconsistent plan, hurting delay and stop scores.

Targeted fixes:

1. Raise the admissible cycle-length range so that the lower bound can never violate the minimum-green requirement.
2. After the Webster calculation, explicitly check and, if necessary, lengthen the cycle so that the effective green time is at least the mandatory minimum (70 s).

This yields a self-consistent timing plan and removes the wasted lost time that previously inflated delays.

Diff 1 – adjust the cycle-length range.

```

<<<<<<< SEARCH
  cycle_length_min, cycle_length_max = 50, 90 # Shorter cycle lengths to reduce delays
=====
  # Increase cycle length bounds to satisfy minimum greens and offer flexibility
  cycle_length_min, cycle_length_max = 90, 140 # Updated bounds
>>>>>>> REPLACE

```

Diff 2 – guarantee feasibility after computing the preliminary cycle length.

```

<<<<<<< SEARCH
  effective_green_time = cycle_length - lost_time
=====
  effective_green_time = cycle_length - lost_time
  # Ensure the cycle length can accommodate all mandatory minimum greens
  min_green_total = 2 * MIN_GREEN_THROUGH + 2 * MIN_GREEN_LEFT
  if effective_green_time < min_green_total:
    cycle_length = min(cycle_length_max, min_green_total + lost_time)
    effective_green_time = cycle_length - lost_time
>>>>>>> REPLACE

```

These two small, consistent edits restore internal feasibility, prevent accidental under-supply of green time, and should improve all three performance metrics.

Fig. 13: Example LLM response introducing the MGF modification

[LLM Response]

I'll select the "**Traffic engineering principles**" strategy category and propose two targeted improvements focused on optimizing the critical movement analysis and green time allocation.

1. First, let's improve the saturation flow adjustment logic to better handle the three evaluation scenarios by making the adjustments more sensitive to demand imbalances:

```
<<<<<<< SEARCH
# Adjust saturation flows based on demand imbalance - higher flows for dominant movements
if total_through_demand > 2 * total_left_demand: # Through-heavy scenario
    sat_flow_through, sat_flow_left = 1900, 1600 # Boost through capacity
elif total_left_demand > 0.6 * total_through_demand: # Left-turn heavy scenario
    sat_flow_through, sat_flow_left = 1800, 1700 # Boost left-turn capacity
else: # Balanced scenario
    sat_flow_through, sat_flow_left = 1850, 1650
=====
# Adjust saturation flows based on demand imbalance with finer granularity
through_left_ratio = total_through_demand / max(1, total_left_demand)
if through_left_ratio > 3.0: # Very through-heavy (Scenario 2)
    sat_flow_through, sat_flow_left = 1950, 1550 # Max through capacity
elif through_left_ratio > 2.0: # Through-heavy
    sat_flow_through, sat_flow_left = 1900, 1600
elif through_left_ratio < 0.8: # Left-turn heavy (Scenario 3)
    sat_flow_through, sat_flow_left = 1750, 1750 # Equal capacity
elif through_left_ratio < 1.2: # Balanced (Scenario 1)
    sat_flow_through, sat_flow_left = 1850, 1700 # Slight left boost
else: # Moderately balanced
    sat_flow_through, sat_flow_left = 1850, 1650
>>>>>>> REPLACE
```

2. Next, let's improve the green time allocation to better prioritize movements that are both critical and high-demand:

```
<<<<<<< SEARCH
# Add small bonus for through movements to reduce stops
demand_weight = y / remaining_demand
if 'through' in phase:
    demand_weight *= 1.2 # 20% bonus for through movements to further reduce stops
extra_g = round(demand_weight * remaining_green)
=====
# Prioritize both critical movements and high absolute demand
demand_weight = y / remaining_demand
if 'through' in phase:
    # Scale bonus based on absolute demand (veh/h) to better handle heavy flows
    phase_demand = sum(traffic_flows.get(f'{d}_through', 0) for d in ['N','S'] if phase.startswith('NS')) or \
        sum(traffic_flows.get(f'{d}_through', 0) for d in ['E','W'] if phase.startswith('EW'))
    demand_weight *= max(1.1, min(1.5, 1 + phase_demand/2000)) # 10-50% bonus scaling with demand
elif critical_flow_ratio[phase] > 0.15: # Only boost important left turns
    demand_weight *= 1.1
extra_g = round(demand_weight * remaining_green)
>>>>>>> REPLACE
```

These changes work together to:

1. More precisely adjust saturation flows based on the actual demand ratios, better handling the three evaluation scenarios. The new thresholds and values better differentiate between the cases we need to handle.
2. Improve green time allocation by:
 - Scaling through movement bonuses based on absolute demand (not just ratios)
 - Only giving left turns a boost when they're truly critical (>15% flow ratio)
 - Capping the maximum bonus to prevent over-allocation

The changes maintain the core algorithm while making it more responsive to the specific traffic patterns we need to handle. The demand-based scaling should particularly help with Scenario 2 (heavy NS) and Scenario 3 (heavy EW+NS left) cases.

Fig. 14: Examples of discarded modifications observed during evolution

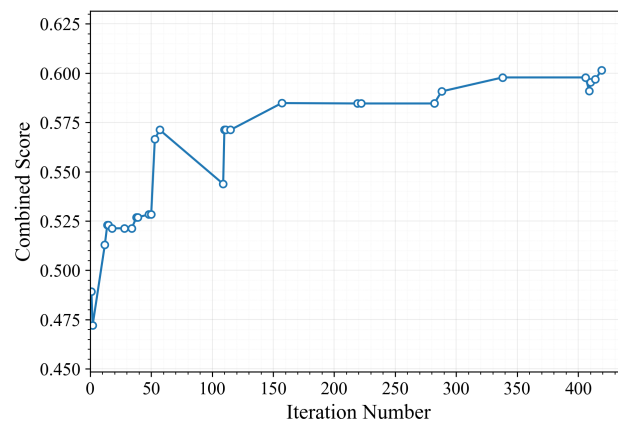


Fig. 15: Evolution path of the best discovered program in a 600-iteration run (alternative experiment)