AGENTICGNN: A SPECTRAL-INSPIRED AGENTIC FRAMEWORK FOR AUTOMATED GRAPH LEARNING ALGORITHM DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

Existing large language model (LLM)-based frameworks for automated graph learning algorithm design typically rely on static performance feedback to conduct black-box iterative optimization. Consequently, they fail to establish explicit associations between target-graph features and algorithm design principle, leading to suboptimal performance and poor cross-task knowledge transfer. Inspired by spectral graph theory, which reveals the intrinsic filtering behaviors of graph learning operators via spectral energy changes, we propose **AgenticGNN**, an agentic framework for automated graph learning algorithm design that integrates multi-level graph feature perception, adaptive spectral-theory-guided context construction, and spectral-aware evolutionary feedback. AgenticGNN first quantifies target-graph characteristics at the neighborhood, community, and global-topology levels, and translates these measurements into an adaptive natural-language context informed by spectral graph theory. This context guides an LLM to generate candidate graph learning algorithms through structured reasoning. During iterative evolution, AgenticGNN analyzes spectral changes to attribute the effects of design decisions on model behavior. The resulting graph features, design rationales, generated code, and evaluation outcomes are distilled into a progressively evolving knowledge base, enabling explicit and reusable mappings from graph characteristics to algorithm design strategies. Experimental results on nine real-world datasets demonstrate that AgenticGNN consistently outperforms state-of-the-art automated design baselines, achieving superior task performance, higher design efficiency, and robust cross-task generalization. The source code is available at https://anonymous.4open.science/r/AgenticGNN-FDCC.

1 INTRODUCTION

Graph structures are considered a natural and efficient way to model real-world complex systems such as social networks, biological networks and transportation systems (Velickovic, 2023), due to their ability to intuitively represent complex relationships between entities (Wu et al., 2021). The continuous development of graph neural networks (GNNs) has been driven by the rapid growth of graph data, along with increasing demand for graph data analysis and mining (Khemani et al., 2024). However, the core architecture of traditional graph neural networks heavily relies on manual design (Zhou et al., 2020). Although graph neural architecture search (GNAS) methods (Gao et al., 2020) achieve automated design via techniques such as reinforcement learning, evolutionary algorithms, or gradient-based search (Gao et al., 2019; Li & King, 2020; Zhao et al., 2021; Wei et al., 2022; Shi et al., 2022; Liu & Liu, 2023; Wang et al., 2025a), they remain fundamentally combinatorial optimization problems bounded by human-defined operator spaces. Consequently, both the search space and optimization strategies still rely on manual configuration (Wang et al., 2023).

In recent years, two general paradigms with higher levels of automation have emerged in graph learning: graph foundation models (GFMs) (Liu et al., 2025) and LLM-based automated algorithm design for graph learning (Wang et al., 2023; Mo et al., 2025). GFMs learn unified graph representations through large-scale pre-training. They have made significant progress in cross-task transfer, cross-domain generalization, and open graph learning, thereby alleviating the reliance of traditional graph neural networks on task-specific architecture design. However, existing GFMs face critical

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

bottlenecks such as the scarcity of large-scale cross-domain datasets, the complexity of unifying diverse graph types and the absence of a standardized pre-training paradigm (Liu et al., 2025). In contrast to directly transferring pretrained capabilities to a new task, automated algorithm design aims to discover task-specific graph learning mechanisms. For LLM-based automated algorithm design methods, they leverage the logical reasoning and code generation capabilities of LLMs (Mo et al., 2025) and transcends the limitations of predefined search spaces in traditional GNAS, enabling the dynamic generation of graph learning algorithms. However, existing LLM-based methods lack systematic modeling and utilization of the characteristics of target graph data during their design process. Specifically, LLMs cannot perceive key properties such as homophily, heterophily, topological structure, and spectral features of graph data when designing algorithms, thereby relying solely on heuristic exploration based on general knowledge. This inevitably leads to a mismatch between candidate structures and graph characteristics, resulting in low search efficiency and cold start problems. Furthermore, existing methods rely on human-engineered heuristics, lacking a mechanism for associating graph data features with graph neural network design principles through automated iterative learning. This not only renders the algorithm design process uninterpretable, but also limits the transferability of knowledge across different tasks and datasets (Liu et al., 2026).

To address these limitations, we propose AgenticGNN, a LLM-based agentic framework that systematically bridges target-graph features with algorithm design strategies via spectral graph theory. Rather than directly applying a fixed pretrained model, AgenticGNN focuses on automatically discovering graph-specific propagation mechanisms through iterative generation, training, and evaluation. While GFMs capture graph-algorithm relationships in latent spaces, AgenticGNN leverages spectral energy dynamics—specifically how operators manipulate spectral components—as an explicit anchor, combining multi-level graph feature perception with spectral reflection to steer automated algorithm design. The main contributions of this paper are summarized as follows:

(1) We introduce a feature perception module that quantifies target-graph characteristics across three distinct levels: micro-neighborhood, community structure, and global topology, providing principled guidelines for the design of automated graph learning algorithms and alleviating the cold start problem in existing methods. (2) We quantify spectral energy changes before and after applying graph learning operators to the target graph as dynamic feedback, revealing the mechanistic filtering behaviors of operators in the spectral domain. (3) A new agentic evolution framework for graph learning algorithm design is designed. Within this framework, graph features, design insights, implementation code, and evaluation results are uniformly stored in a short-term population memory and a continuously evolving knowledge base. By combining these historical experiences with performance feedback, LLMs can infer the causal links between design decisions and model behaviors. This continuously refines better mapping from graph characteristics to algorithm, ultimately achieving highly efficient, interpretable, and transferable algorithm generation.

2 Related Work

In this section, we briefly review three related research directions. A more detailed review is deferred to Appendix A.

Spectral Graph Neural Networks. Spectral Graph Neural Networks (SGNNs) capture underlying graph patterns by designing spectral filters over graph Laplacians. Early works such as ChebNet (Defferrard et al., 2016) and GCN (Kipf & Welling, 2017) approximate filters via polynomials, laying the foundation for adaptive polynomial-based filters (e.g., GPRGNN (Chien et al., 2021), BernNet (He et al., 2021)), spatial-spectral combinations (Xu et al., 2024), and message-passing designs (Yue et al., 2025; Zheng et al., 2024).

Graph Neural Architecture Search. GNAS aims to automate GNN architecture design and hyperparameter selection. Early methods explore predefined spaces using reinforcement learning (Gao et al., 2019), genetic algorithms (Li & King, 2020; Shi et al., 2022), or gradient-based differentiable search (Zhao et al., 2021). Subsequent works expand the search space to topological connections and feature fusion (Wei et al., 2022; Liu & Liu, 2023), while recent advances leverage external knowledge bases for architecture generation (Wang et al., 2025a). However, GNAS remains essentially a combinatorial optimization problem constrained by expert-defined operators.

Automated Algorithm Design. In the field of automated algorithm design, FunSearch (Romera-Paredes et al., 2024) combines LLMs with evolutionary methods to successfully discover new mathematical constructions; EoH (Liu et al., 2024) uses Chain-of-Thought (CoT) to guide LLMs in

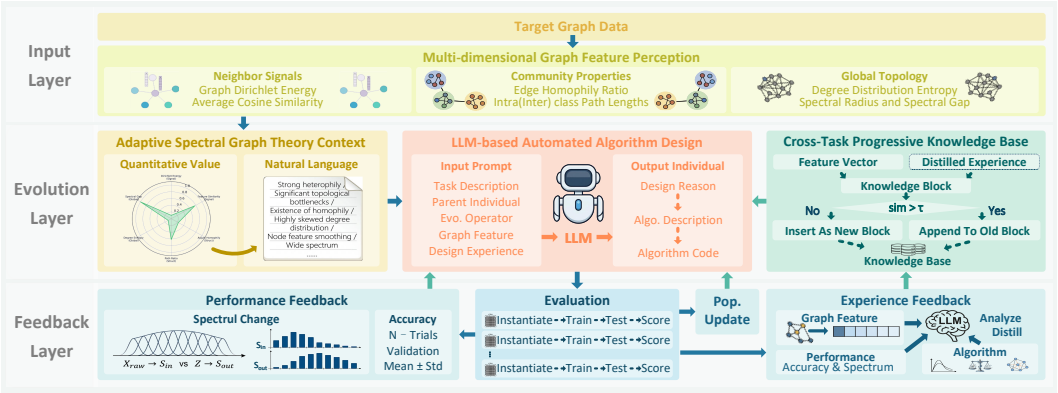


Figure 1: The overview of AgenticGNN.

mutating and crossing algorithm code, thereby accelerating the search process of heuristic algorithms. LLM-SR (Shojaee et al., 2025) utilises LLMs to automatically discover potential scientific equations and physical laws. GNAS-LLM (Wang et al., 2025b) combines GNNs with LLMs for the first time; AutoSGNN (Mo et al., 2025) constructs a unified SGNN search space, utilising LLMs to generate spectral graph filter algorithm code adapted to specific datasets through evolutionary iteration. AlphaEvolve (Novikov et al., 2025) enhances capabilities of state-of-the-art LLMs on highly challenging tasks as an evolutionary coding agent.

3 METHODOLOGY

In this section, we introduce the proposed framework, termed AgenticGNN, for automated graph learning algorithm design. This framework introduces a context-construction mechanism that integrates target graph features with spectral graph theory, and employs an evolutionary generation strategy based on design rationale and CoT reasoning. As illustrated in Figure 1, AgenticGNN consists of five parts: a graph feature perception module, an adaptive spectral graph theory-enhanced context module, a dynamic performance feedback module, a knowledge base module for cross-task continuously evolving graph learning, and an LLM-based automated graph learning algorithm generation module.

3.1 MULTI-DIMENSIONAL GRAPH FEATURE PERCEPTION

This module extracts key neighborhood, community and global topological features of target graph, guiding the design of core algorithm components. The Dirichlet energy of the graph perceive the smoothness of neighborhoods in the graph data, reflecting the proportion of high and low frequency components in the graph data. For a graph $G = (V, E)$, the Dirichlet energy ϵ_{dir} is expressed as:

$$\epsilon_{\text{dir}} = \frac{1}{2} \sum_{(i,j)\in E} a_{ij} \left\| \frac{x_i}{\sqrt{1+d_i}} - \frac{x_j}{\sqrt{1+d_j}} \right\|_2^2 \tag{1}$$

where a_{ij} is the element in the adjacency matrix A, x_i is the feature vector of node i, and d_i is the degree of node i. The average feature cosine similarity of nodes reflects the feature similarity between adjacent nodes in the graph:

$$S_{\text{feat}} = \frac{1}{|E|} \sum_{(i,j)\in E} \frac{x_i^T x_j}{\|x_i\|_2 \|x_j\|_2} \tag{2}$$

where $|E|$ denotes the total number of edges, and $\|x_i\|_2$ denotes the L_2-norm of the feature vector x_i. The edge homogeneity ratio reflects the degree of homogeneity in the graph via the proportion of edges connecting nodes with the same class label:

$$H_{\text{edge}} = \frac{1}{|E|} \sum_{(i,j)\in E} \mathbb{1}(y_i = y_j) \tag{3}$$

where $\mathbb{1}(\cdot)$ is the indicator function that equals 1 if the condition holds and 0 otherwise, and y_i represents the class label of node i. It is calculated using only edges within the training set to simulate the sparsity of class labels found in real-world graph data. To eliminate biases arising from differences in category label distribution and node degrees, we further calculate the expected edge homogeneity under the assumption of randomly connected edges:

$$E[H] = \sum_{k=1}^{K} \left(\frac{D_k}{2|E|} \right)^2 \tag{4}$$

where k is the category index, K is the total number of categories, and D_k is the total degree of all nodes labelled k. The adjusted edge homogeneity ratio is thus obtained as follows:

$$H_{adjusted} = \frac{H_{edge} - E[H]}{1 - E[H]} \tag{5}$$

The average intra-class path length and the average inter-class path length reflect the clustering characteristics of categories within communities:

$$L_{\text{intra}} = \frac{1}{K} \sum_{k=1}^{K} \frac{2}{|C_k|(|C_k| - 1)} \sum_{i,j \in C_k, i \neq j} d_{ij} \tag{6}$$

$$L_{\text{inter}} = \frac{1}{\binom{K}{2}} \sum_{1 \leq p < q \leq K} \frac{1}{|C_p||C_q|} \sum_{i \in C_p, j \in C_q} d_{ij} \tag{7}$$

where K denotes the total number of classes, C_k represents the set of nodes belonging to class k, and d_{ij} denotes the shortest path length between node i and node j. Degree distribution entropy reflects the disorder and diversity of the graph's topological structure:

$$p_k = \frac{|\{i \in V \mid d_i = k\}|}{N}, H_{\text{deg}} = -\sum_{k} p_k \log p_k \tag{8}$$

where $N = |V|$ is the total number of nodes, and p_k is the probability of a node having degree k. The spectral radius and spectral gap reflect the global connectivity and bottlenecks:

$$\rho = \max \lambda(L) = \lambda_n, \Delta\lambda = \lambda_2 - \lambda_1 = \lambda_2 \tag{9}$$

where $\lambda(L)$ denotes the eigenvalues of the normalized Laplacian matrix L, sorted in ascending order $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. In classical spectral graph theory, as governed by Cheeger's inequality, the second-smallest eigenvalue λ_2 explicitly characterizes the topological bottlenecks and global connectivity of the graph. A smaller $\Delta\lambda$ indicates a graph with distinct, tightly-knit communities separated by sparse bottlenecks. To avoid the enormous computational overhead of a full eigenvalue decomposition, since the time complexity is $\mathcal{O}(N^3)$, we employ the implicitly restarted Lanczos method (Sorensen, 1992) to efficiently approximate these extreme eigenvalues of the normalized Laplacian matrix. This strictly controls the computational complexity to $\mathcal{O}(k \cdot |E|)$, where the number of iterations $k \ll N$.

3.2 Adaptive Spectral Graph Context

Raw numerical graph feature metrics lack explicit semantic meanings, preventing LLMs from comprehending their implications for algorithm design rationales. To mitigate this barrier, we construct an adaptive, spectral-theory-guided context. For a target graph, if matching historical experiences exist in the cross-task knowledge base (detailed in Section 3.4), AgenticGNN utilizes them to construct the context prompt. Conversely, if no eligible experiences are available, AgenticGNN map quantitative graph metrics onto qualitative natural language descriptions via rule-based thresholds.

3.3 Performance Feedback

Spectral change feedback aims to characterise the influence of graph learning operators on target graph signal in spectral domain by analysing changes of spectral energy distribution before and after the operator is applied to the target graph data, thereby guiding the optimization of graph learning

operator. Given a target graph $G = (V, E)$, it's symmetric normalised Laplacian matrix is defined as: $L = I_n - D^{-1/2}AD^{-1/2}$. Since the computational complexity of performing an eigenvalue decomposition of L is $\mathcal{O}(|V|^3)$, which is infeasible, this paper employs an adaptive Gaussian filter bank based on Chebyshev polynomial approximation to capture the spectral energy density of the graph signal. The theoretical interval $[0, 2]$ of the eigenvalues of the Laplace matrix is uniformly divided into $M = 10$ sub-bands. For the m-th frequency band, the Gaussian bandpass filter function $g_m(\lambda)$ is defined as:

$$g_m(\lambda) = \exp\left(-\frac{(\lambda - \mu_m)^2}{2\sigma^2}\right) \tag{10}$$

where λ is the eigenvalue of the Laplace matrix, μ_m is the centre frequency of the band, and σ is the bandwidth controlling the degree of band overlap. Defining the shift-normalised Laplace matrix as $\tilde{L} = L - I_n$, we have $\tilde{\lambda} = \lambda - 1$, which maps $g_m(\lambda)$ to the domain of Chebyshev polynomials $[-1, 1]$. The filter operator $g_m(L)$ can be approximated using a K-th-order Chebyshev expansion:

$$g_m(L)X \approx \sum_{k=0}^{K} c_{m,k} T_k(\tilde{L})X \tag{11}$$

where X is the input graph signal, $c_{m,k}$ are the Chebyshev expansion coefficients, and $T_k(\tilde{L})X$ can be obtained via a recursive relation in $\mathcal{O}(|E|)$ linear complexity:

$$T_0(\tilde{L})X = X, T_1(\tilde{L})X = \tilde{L}X, T_k(\tilde{L})X = 2\tilde{L}T_{k-1}(\tilde{L})X - T_{k-2}(\tilde{L})X \tag{12}$$

The input spectrum S_{in} is obtained by passing the raw node features X_{raw} through a Gaussian filter bank, and is used to characterise the intrinsic spectral density distribution of the raw data. For the m-th bandpass frequency band, the energy proportion $S_{\text{in}}(m)$ is given by:

$$S_{\text{in}}(m) = \frac{\|g_m(\tilde{L})X_{\text{raw}}\|_F^2}{\sum_{j=1}^{M} \|g_j(\tilde{L})X_{\text{raw}}\|_F^2} \tag{13}$$

The output spectrum S_{out} is obtained by passing the hidden-layer features Z—which precede the classifier head—through a Gaussian filter bank; it characterises the spectral density distribution of the graph signal following the application of the graph neural network to the graph data. For the m-th bandpass frequency band, the energy proportion $S_{\text{out}}(m)$ is given by:

$$S_{\text{out}}(m) = \frac{\|g_m(\tilde{L})Z\|_F^2}{\sum_{j=1}^{M} \|g_j(\tilde{L})Z\|_F^2}, Z \in \mathbf{R}^{|V| \times d_{\text{hidden}}} \tag{14}$$

As operators generated by random mutations in large models are highly prone to introducing unreasonable matrix multiplications or non-linear activations—leading to the divergence of the Chebyshev recursion and feature space collapse—this module incorporates state anomaly feedback. During evaluation, if the output feature tensor satisfies $\exists z_{ij} \in Z$ s.t. $z_{ij} \in \{\text{NaN}, \pm\infty\}$, or the recursion norm overflows such that $\|T_k(\tilde{L})X\|_F > \tau$ ($\tau = 10^5$), the spectral matrix of that individual is deemed to be in a state of numerical collapse, and its spectral feedback information is discretised into: $S_{\text{out}} \leftarrow \Phi_{\text{error}} = [-1.0, -1.0, \ldots, -1.0]^T \in \mathbf{R}^M$.

3.4 Cross-Task Progressive Knowledge Base

Existing LLM-based methods in graph learning lack a long-term memory mechanism to continuously accumulate cross-task experience for inspiring algorithm designs. This paper constructs a cross-task progressive knowledge base, capturing the underlying mapping between graph features and graph learning algorithms, thereby imparting transferable design experience for subsequent tasks. Specifically, each knowledge block is formulated as a tuple containing a graph feature vector and its corresponding design experiences described by natural language, which can be represented as: $K_i = (\mathcal{F}_i, \mathcal{L}_i)$. $\mathcal{F}_i$ is the graph feature vector obtained from the graph feature index system: $\mathcal{F}_i = \{\mathbf{f}_{i,1}, \mathbf{f}_{i,2}, \ldots, \mathbf{f}_{i,N_i}\}$, $\mathcal{L}_i$ is a set of corresponding algorithm design experience: $\mathcal{L}_i = \{\mathcal{E}_{i,1}, \mathcal{E}_{i,2}, \ldots, \mathcal{E}_{i,N_i}\}$. At the end of each evolutionary generation, LLMs distill algorithm design experience by extracting insights from target graph features, algorithms and dynamic

performance feedback of newly evolved individuals. Subsequently, if the maximum similarity between normalized feature vector of target graph and existing knowledge block exceeds threshold τ, the newly distilled experience and new graph feature vector will be updated to the corresponding knowledge block. Otherwise, the knowledge tuple will be appended to the knowledge base as a novel block. When the number of experiences within a block exceeds a predefined limit, the LLM is prompted to consolidate them into a unified, coherent summary. Additionally, a global capacity constraint is enforced by retaining only the most recently updated blocks to prevent storage overflow.

$$\mathcal{K} \leftarrow \begin{cases} \text{Update}(K_i, \mathcal{E}_{\text{new}}), & \text{Sim}(\mathbf{f}_{\text{new}}, \mathbf{f}_i) > \tau, \\ \mathcal{K} \cup \{K_{\text{new}}\}, & \text{otherwise.} \end{cases} , \text{Sim}(\mathbf{f}_i, \mathbf{f}_j) = \frac{\mathbf{f}_i^\top \mathbf{f}_j}{\|\mathbf{f}_i\|_2 \, \|\mathbf{f}_j\|_2} \tag{15}$$

For a new graph learning task or iteration, we first retrieve the top-K most similar historical knowledge block from the knowledge base, serving as context to guide LLMs in designing algorithms. This mechanism facilitates the reuse of algorithm design experience across diverse tasks.

3.5 LLM-based Automated Algorithm Design

This module is implemented within an evolutionary search framework, comprising an elite selection strategy, a mechanism for generating candidate individuals, and a parallel evaluation process. It enable LLMs to generate graph learning algorithms with clear design attribution by combining target graph feature, spectral graph theory, design experience with CoT reasoning.

In this module, we define three evolutionary prompt strategies—mutation, crossover and direct preference optimisation(DPO)—and enhances these strategies with context-aware mechanisms, allowing LLMs to autonomously derive evolutionary directions by utilising the graph feature metric system, task preformance feedback and the design rationale of parent elite individuals.

In AgenticGNN, we continuously maintain a fixed number of evolutionary populations. Each individual consists of a quadruple of code, spectral changes, design philosophy, and target score, which serves as historical context to fill in the prompts for evolutionary operations, achieving optimization with explicit logical chains. The evolutionary objective function is defined as: $\text{objective} = \mu_{\text{acc}} - \lambda\sigma_{\text{acc}}$. Within each evolutionary iteration, LLMs first outputs a design attribution analysis of graph learning algorithm based on graph data features, spectral changes and task performance feedback. Then, outputs specific implementation approaches and generate *GNN_Layer* class code based on these approaches. The automated design therefore focuses on the graph convolutional operators, while the overall backbone structure and training procedure are kept fixed to provide a controlled search space. The GNN models obtained by instantiating individual code instances are distributed to a GPU cluster for parallel evaluation. To filter out invalid or inefficient architectures, a resource-constrained strategy is adopted, incorporating an early-stopping mechanism to prevent overfitting, as well as a timeout cut-off. If a model fails to converge or times out during training, the evaluation process is terminated and the individual is marked as eliminated.

4 Experiments

4.1 Experimental Setup

Dataset. Nine widely-used graph benchmark datasets are utilized for the experiments. Homogeneous graphs included *Cora*, *CiteSeer*, *PubMed*, *Computers* and *Photo*. Heterogeneous graphs included *Chameleon*, *Squirrel*, *Texas* and *Cornell*. Details of datasets are shown in Appendix B.
Baseline. We compare AgenticGNN with four categories of representative baselines: 1) artificially designed SGNNs including **GCN** (Kipf & Welling, 2017), **ChebNet** (Defferrard et al., 2016), **APPNP** (Klicpera et al., 2019), **GPRGNN** (Chien et al., 2021), **FAGCN** (Bo et al., 2021), **BernNet** (He et al., 2021), **JacobiConv** (Wang & Zhang, 2022) and **NFGNN** (Zheng et al., 2024), 2) GNAS methods including **CTFGNAS** (Liu & Liu, 2023), **F2GNN** (Wei et al., 2022), **Genetic-GNN** (Shi et al., 2022) and **SANE** (Zhao et al., 2021), 3) LLM-based method including **AutoSGNN** (Mo et al., 2025) and **GNAS-LLM** (Wang et al., 2025b) and 4) adaptive GFMs represented by **GraphPFN** (Eremeev et al., 2025) and **TFSGFM** (Wang et al., 2026). Experiments on manually designed SGNN benchmark followed the experimental setup of NFGNN (Zheng et al., 2024). Experiments on GNAS benchmark followed the experimental setup of CTFGNAS (Liu & Liu, 2023). Experiments on AutoSGNN, GNAS-LLM, GraphPFN and TFSGFM followed the optimal configurations in the

Table 1: Accuracy of node-classification task (2.5% : 2.5% : 95%) on nine benchmark datasets with the best results highlighted in bold and the second-best results underlined.

Model	*Cora*	*CiteSeer*	*PubMed*	*Computers*	*Photo*	*Chameleon*	*Squirrel*	*Texas*	*Cornell*	Rank
GCN	75.21 ± 0.38	67.30 ± 0.35	84.27 ± 0.01	82.52 ± 0.32	90.54 ± 0.21	35.73 ± 0.87	28.37 ± 0.77	32.40 ± 2.31	35.90 ± 3.64	10
ChebNet	71.39 ± 0.51	65.67 ± 0.38	83.83 ± 0.12	82.41 ± 0.28	90.09 ± 0.28	32.36 ± 1.61	27.76 ± 0.93	28.51 ± 4.03	26.21 ± 3.12	14
APPNP	79.41 ± 0.38	**68.59 ± 0.30**	85.02 ± 0.09	81.99 ± 0.26	91.11 ± 0.26	30.12 ± 0.92	26.01 ± 0.48	46.44 ± 3.10	44.37 ± 4.90	8
GPRGNN	79.51 ± 0.36	67.63 ± 0.38	85.07 ± 0.09	82.90 ± 0.37	91.93 ± 0.26	31.46 ± 0.89	26.14 ± 0.51	44.76 ± 4.33	43.45 ± 4.65	6
FAGCN	78.10 ± 0.21	66.77 ± 0.18	84.09 ± 0.02	82.11 ± 1.55	90.39 ± 1.34	37.24 ± 3.54	26.91 ± 3.08	48.44 ± 1.78	46.38 ± 1.82	7
BernNet	76.37 ± 0.36	65.83 ± 0.61	82.57 ± 0.17	79.57 ± 0.28	91.60 ± 0.35	27.33 ± 1.14	26.42 ± 0.39	48.21 ± 3.17	46.50 ± 5.57	11
JacobiConv	75.30 ± 0.02	62.75 ± 0.02	85.05 ± 0.01	**87.89 ± 0.01**	92.34 ± 0.01	39.13 ± 0.02	25.91 ± 0.01	56.34 ± 0.05	44.80 ± 0.04	4
NFGNN	77.69 ± 0.91	67.74 ± 0.52	85.07 ± 0.13	84.18 ± 0.40	92.16 ± 0.82	40.10 ± 1.34	30.91 ± 0.62	50.37 ± 6.54	48.37 ± 7.34	3
CTFGNAS	75.72 ± 2.06	64.24 ± 1.85	83.35 ± 1.41	84.98 ± 1.41	90.52 ± 0.96	37.54 ± 5.84	31.47 ± 1.73	57.03 ± 3.47	37.38 ± 2.23	5
F2GNN	65.33 ± 1.26	58.16 ± 2.28	81.16 ± 0.69	46.81 ± 4.01	82.03 ± 3.13	35.72 ± 1.86	24.52 ± 1.41	50.06 ± 4.47	48.44 ± 7.58	16
Genetic-GNN	76.58 ± 1.06	60.98 ± 1.67	84.01 ± 0.46	83.30 ± 2.10	87.33 ± 1.37	38.54 ± 2.00	29.81 ± 1.09	44.45 ± 6.34	45.74 ± 7.87	9
SANE	73.52 ± 3.58	61.57 ± 2.62	82.91 ± 0.26	80.41 ± 2.15	87.56 ± 1.70	41.79 ± 2.33	28.79 ± 1.09	49.71 ± 2.92	35.61 ± 13.32	12
GraphPFN	60.10 ± 1.11	47.41 ± 3.78	**87.04 ± 0.39**	74.70 ± 1.70	85.90 ± 0.94	38.81 ± 1.87	31.23 ± 0.39	39.02 ± 16.07	24.46 ± 11.75	13
TFSGFM	68.66 ± 3.62	66.21 ± 1.15	68.81 ± 0.69	72.55 ± 1.48	77.18 ± 4.77	29.30 ± 3.27	22.12 ± 1.37	34.86 ± 19.88	28.23 ± 15.66	17
GNAS-LLM	55.30 ± 0.25	52.93 ± 0.22	81.02 ± 0.06	81.89 ± 0.06	89.53 ± 0.06	35.76 ± 0.19	25.28 ± 0.14	55.66 ± 0.69	44.74 ± 1.06	15
AutoSGNN	80.61 ± 1.52	67.23 ± 1.31	85.13 ± 0.26	84.80 ± 0.82	**92.51 ± 0.36**	38.67 ± 3.52	31.17 ± 1.82	57.10 ± 4.86	50.87 ± 6.50	2
AgenticGNN	**80.64 ± 1.40**	67.91 ± 2.19	85.17 ± 0.24	86.29 ± 0.72	92.50 ± 0.28	**43.62 ± 3.43**	**32.99 ± 1.57**	**62.83 ± 3.12**	**53.99 ± 9.05**	1

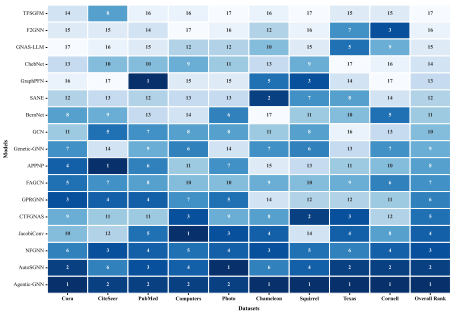


Figure 2: Rank comparison heatmap.

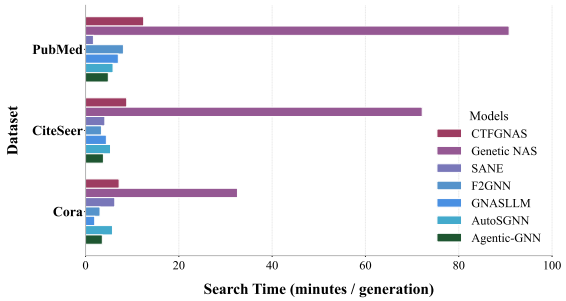


Figure 3: Model's search time per generation.

respective papers. The best hyperparameters provided in papers were used for each dataset.

Reproducibility. All experiments in this study (except GraphPFN, which was conducted on an H200) were carried out on a platform equipped with 4 NVIDIA GeForce RTX 3090 GPUs, 2 Intel Xeon Silver 4210 CPUs and 252GB of memory. The GPT-4o pre-trained model was selected as the algorithm's generation core with 4 parallel responses to enhance generation efficiency whilst ensuring diversity. Experiments are configured with 20 iterations (12 candidates per round, population size 30) using a unified search space of 2 layers of automatically generated convolutional layers and 2 layers of MLP layers for feature transformation. Training adopts early stopping at 200 epochs with a 600-second timeout. All experiments were independently repeated 10 times, yielding the mean and standard deviation of the accuracy.

4.2 Experimental Results

Node-Classification Task. We employed a classic sparse partitioning strategy with 2.5% nodes for training, 2.5% for validation, and 95% for testing. Table 1 lists the results on node-classification task. AgenticGNN achieved the highest accuracy on five datasets and the second-highest accuracy on four datasets. Furthermore, it improved accuracy by approximately 5% over the previous best method on the Chameleon and Texas node classification tasks, and by approximately 3% on the Cornell node classification task. After Wilcoxon signed-rank test and Holm correction (Fawaz et al., 2019), at a significance level of p=0.05,AgenticGNN ranks first among 17 frameworks across all datasets. The rank heatmap is shown in Figure 9.

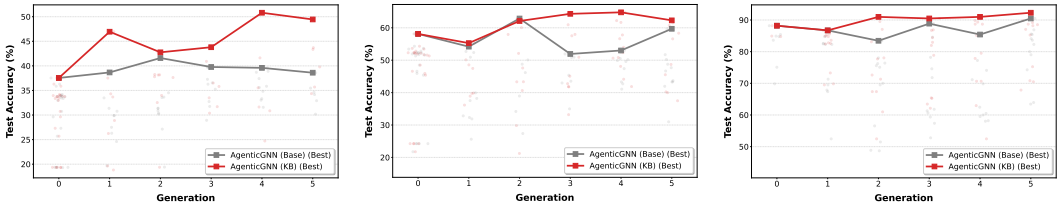
Efficiency and Performance. As shown in Figure 3, the average execution time of AgenticGNN is on the same magnitude as other methods, and the accuracy of the algorithm obtained by AgenticGNN after only 20 rounds iterations has exceeded the accuracy of the algorithm obtained by the previous model after 30 rounds, which greatly improves the efficiency and performance of automated algorithm generation.Regarding the results of the node classification tasks on *Squirrel*, *Chameleon*

Table 2: Ablation on node classification.

Model	*Computers*	*Chameleon*	*Cornell*
AutoSGNN	84.80±0.82	38.67±3.52	50.87±6.50
AgenticGNN(v1)	85.03±0.99	39.67±4.73	51.45±8.90
AgenticGNN(v2)	**86.29±0.72**	**43.62±3.43**	**53.99±9.05**

Table 3: Fine-grained ablation on *Squirrel*.

Feature Metrics	Qualitative Mapping	Variance Penalty	Accuracy (%)
✓	×	×	31.94 ± 1.10
✓	✓	×	32.19 ± 2.04
✓	✓	✓	**32.45 ± 0.72**



(a) Squirrel (b) Chameleon (c) Cornell

Figure 4: Accuracy curve of node-classification task (5 iterations).

and *Cornell*, the accuracy curve in Figure 4 demonstrate that AgenticGNN exhibits substantial advantages in search efficiency and performance, and has overcome the performance bottleneck on heterogeneous graphs.Time complexity analysis is provided in Appendix C and D.

Ablation Study. To validate the effectiveness of graph feature metrics and spectral change feedback, we conduct ablation experiments on *Computers*, *Chameleon* and *Cornell*. AgenticGNN(v1) introduces graph feature metrics based on AutoSGNN, while AgenticGNN(v2) introduces spectral change feedback based on AgenticGNN(v1).Table 2 presents the node-classification accuracy. Results show that both graph feature metrics and spectral change feedback enhance the performance of LLM-based automated graph learning algorithm design. To further dissect the fine-grained contribution, we present an ablation in Table 3 on node-classification task with the split ratio of $2.5\% : 2.5\% : 95\%$ and iterations of 20, showing that each subtle component contributes to performance gains. Building upon AgenticGNN(v2) (denoted as Base), we conduct an ablation study under node-classification tasks with a dense split ($60\% : 20\% : 20\%$) across 5 iterations to quantitatively isolate the performance gains yielded by the global cross-task Knowledge Base (KB). The ablation results are shown in Table 4 and an accuracy curve on *Squirrel* is shown in Figure 4a. Furthermore, we perform additional ablations with the same dense split under a zero-shot setting (i.e., without population initialization), demonstrating the effectiveness of AgenticGNN in addressing the cold-start problem. The results are reported in Table 5.

5 Discussion

Analyze of Spectral Change Feedback. To unveil the underlying mechanisms of AgenticGNN, we investigate the efficacy and boundary conditions of its autonomous spectral optimization. On highly heterophilic graphs (e.g., *Chameleon*), the initial graph signals exhibit severe energy concentration in mid-to-high frequency bands, matching a high Dirichlet energy. AgenticGNN effectively counteracts this via global spectral debiasing and flattening, redistributing energy to enhance representation discriminability. Conversely, on strongly homophilic configurations (e.g., *PubMed*), where the spectral gap is extremely narrow, high-frequency components natively manifest as structural noise rather than semantic features. Merely spectral micro-tuning disrupts global low-frequency smoothness, leading to marginal gains or even negative optimization. This contrast demonstrates that pure spectral feedback operates under inherent topological boundaries. Crucially, instead of blindly relying on a singular spectral paradigm, AgenticGNN dynamically adapt its evolutionary trajectory toward spatial-domain augmentations base on graph feature perception

Table 4: Ablation of KB on node-classification task (5 iters) with the split ratio of $60\% : 20\% : 20\%$.

Model	*Cora*	*CiteSeer*	*PubMed*	*Computers*	*Photo*	*Chameleon*	*Squirrel*	*Texas*	*Cornell*	Rank
AgenticGNN (Base)	**89.10 ± 0.98**	81.32 ± 1.46	**90.82 ± 0.55**	**90.44 ± 0.50**	94.56 ± 0.39	62.87 ± 2.39	41.59 ± 1.11	90.49 ± 4.07	90.49 ± 3.72	2
AgenticGNN (Single_Task_KB)	89.00 ± 1.03	81.27 ± 1.39	90.65 ± 0.54	89.77 ± 1.01	95.08 ± 0.31	62.71 ± 2.71	49.55 ± 1.58	**91.97 ± 4.78**	90.49 ± 4.20	3
AgenticGNN (Cross_Task_KB)	89.05 ± 1.04	**81.32 ± 1.42**	90.66 ± 0.76	89.92 ± 0.83	**95.14 ± 0.64**	**64.77 ± 3.74**	**50.81 ± 1.58**	91.48 ± 3.34	**92.30 ± 3.88**	1

Table 5: Ablation on zero-shot node-classification task with the split ratio of $60\% : 20\% : 20\%$.

Model	*Cora*	*CiteSeer*	*PubMed*	*Computers*	*Photo*	*Chameleon*	*Squirrel*	*Texas*	*Cornell*
AutoSGNN	87.09 ± 0.95	76.45 ± 1.95	89.31 ± 0.45	$\mathbf{89.90 \pm 0.57}$	93.87 ± 0.47	56.76 ± 8.54	41.25 ± 1.28	$\mathbf{87.38 \pm 5.34}$	85.74 ± 5.54
AgenticGNN (Base)	$\mathbf{87.57 \pm 1.42}$	$\mathbf{77.24 \pm 1.45}$	$\mathbf{90.37 \pm 0.41}$	78.59 ± 9.12	$\mathbf{94.03 \pm 0.58}$	$\mathbf{68.29 \pm 3.30}$	$\mathbf{53.30 \pm 1.61}$	87.21 ± 5.43	$\mathbf{88.52 \pm 4.02}$

Table 6: Intra- and inter-cluster similarities of normalized graph feature vectors.

Category	**Group 1**	**Group 2**	**Similarity Range**	**Mean Similarity**
Intra-Cluster Cohesion	Cluster 1	Cluster 1	0.9865 – 0.9989	**0.9925**
	Cluster 2	Cluster 2	0.9985 – 0.9985	**0.9985**
	Cluster 3	Cluster 3	0.9974 – 0.9974	**0.9974**
Inter-Cluster Separation	Cluster 1	Cluster 3	0.8912 – 0.9544	**0.9224**
	Cluster 2	Cluster 3	0.8798 – 0.9039	**0.8915**
	Cluster 1	Cluster 2	0.8470 – 0.8823	**0.8669**

result. The spectral change curves and detailed case studies are deferred to Appendix D.

Analyze of Cross-Task Progressive Knowledge Base. The storage and retrieval of cross-task progressive knowledge base rely on normalized graph feature vectors. We calculate the pairwise cosine similarity of normalized graph feature vectors across the 9 benchmark datasets and group them into three distinct clusters identified by the PCA. As summarized in Table 6, high intra-cluster cohesion and pronounced inter-cluster separation are observed. The normalization details, similarity values, and cluster distributions are shown in Appendix E. **Interpretability Analysis.** In contrast to black-box automated algorithm design, AgenticGNN exhibits high interpretability. For instance, when optimizing for the heterophilic *Chameleon* dataset, AgenticGNN explicitly designs a hierarchical frequency decomposition layer modulated by signal-controlled band-pass filters, precisely targeting the erratic high-frequency noise and energy migration inherent in heterophily. The generated design reasoning and code for the optimal architectures discovered across all benchmark datasets can be found in Appendix F.

Key Parameter Analysis Although a unified population size of 30 was adopted to align with the base model, experiments reveal that a size of 10 performs better on certain datasets. To quantify the impact of population size, we conduct a sensitivity analysis across multiple datasets, with detailed results shown in Appendix G due to the page limit.

Visualization. To provide deeper insights into the evolutionary process, empirical results, and graph feature spaces, comprehensive visualizations are presented in Appendix H. Specifically, we present accuracy curves for the fine-grained and knowledge-base ablation studies, along with PCA projection and radar plots of the normalized graph feature vectors.

Limitation. While AgenticGNN establishes an explicit link between graph features and algorithm, its effectiveness is more pronounced on heterophilic graphs, while its benefits are comparatively limited on homophilic graphs. Although AgenticGNN enables graph propagation mechanisms to better adapt to the target graph through iterative candidate generation and evaluation, this adaptation incurs additional computational cost compared with directly reusing existing solvers.

6 Conclusion

In this paper, we propose AgenticGNN, an agentic framework for automated graph learning algorithm design. By integrating multi-dimensional feature perception, adaptive spectral context construction, and dynamic spectral-energy feedback, AgenticGNN overcomes the limitations of black-box optimization and provides mechanistic insights into operator behaviors. Crucially, by distilling design rationales and evaluation outcomes into an evolving cross-task knowledge base, the framework establishes explicit and reusable causal mappings from graph features to algorithmic methodologies. Extensive experiments on nine real-world datasets demonstrate that AgenticGNN achieves state-of-the-art accuracy—ranking first among 17 models, delivering superior search efficiency and robust cross-task transferability.

7 Reproducibility Statement

We have made every effort to ensure that the results presented in this paper are reproducible. All code and datasets have been made publicly available in an anonymous repository to facilitate replication and vcrification. The cxperimental sctup, including training steps, modecl configurations, and hardware details, is described in detail in the paper. We have aiso provided a full description of (Your contributiong, to assist others in rcproducing our experiments. Additionally, (Types of public datasels used in the paper,, such as (Specific dataset names), are publicly available, cnsuring consistent and reproducible cvaluation results We believe these measures will enable other researchers to reproduce our work and further advance the ficld.

8 Ethics Statement

This work adheres to the ICLR Code of Ethics. In this study, no human subjccts or animal experimentation was involved. All datasets used were sourced in compliance with relevant usage guidelines, ensuring no violation of privacy. We have taken care to avoid any biases or discriminatory outcomes in our research process. No personally identifiable information was used, and no experiments were conducted that could raise privacy or security concerns. We are committcd to maintaining transparency and intcgrity throughout the rescarch process.

References

Deyu Bo, Xiao Wang, Chuan Shi, and Huawei Shen. Beyond low-frequency information in graph convolutional networks. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pp. 3950–3957. AAAI Press, 2021. doi: 10.1609/AAAI.V35I5.16514. URL https://doi.org/10.1609/aaai.v35i5.16514.

Eli Chien, Jianhao Peng, Pan Li, and Olgica Milenkovic. Adaptive universal generalized pagerank graph neural network. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=n6jl7fLxrP.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 3837–3845, 2016. URL https://proceedings.neurips.cc/paper/2016/hash/04df4d434d481c5bb723be1b6df1ee65-Abstract.html.

Dmitry Eremeev, Oleg Platonov, Gleb Bazhenov, Artem Babenko, and Liudmila Prokhorenkova. Graphpfn: A prior-data fitted graph foundation model. *CoRR*, abs/2509.21489, 2025. doi: 10.48550/ARXIV.2509.21489. URL https://doi.org/10.48550/arXiv.2509.21489.

Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data Min. Knowl. Discov.*, 33(4): 917–963, 2019. doi: 10.1007/S10618-019-00619-1. URL https://doi.org/10.1007/s10618-019-00619-1.

Yang Gao, Hong Yang, Peng Zhang, Chuan Zhou, and Yue Hu. Graphnas: Graph neural architecture search with reinforcement learning. *CoRR*, abs/1904.09981, 2019. URL http://arxiv.org/abs/1904.09981.

Yang Gao, Hong Yang, Peng Zhang, Chuan Zhou, and Yue Hu. Graph neural architecture search. In Christian Bessiere (ed.), *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pp. 1403–1409. ijcai.org, 2020. doi: 10.24963/IJCAI.2020/195. URL https://doi.org/10.24963/ijcai.2020/195.

Mingguo He, Zhewei Wei, Zengfeng Huang, and Hongteng Xu. Bernnet: Learning arbitrary graph spectral filters via bernstein approximation. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 14239–14251, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/76f1cfd7754a6e4fc3281bcccb3d0902-Abstract.html.

Bharti Khemani, Shruti Patil, Ketan Kotecha, and Sudeep Tanwar. A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions. *J. Big Data*, 11(1):18, 2024. doi: 10.1186/S40537-023-00876-4. URL https://doi.org/10.1186/s40537-023-00876-4.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Predict then propagate: Graph neural networks meet personalized pagerank. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=H1gL-2A9Ym.

Yaoman Li and Irwin King. Autograph: Automated graph neural network. In Haiqin Yang, Kitsuchart Pasupa, Andrew Chi-Sing Leung, James T. Kwok, Jonathan H. Chan, and Irwin King (eds.), *Neural Information Processing - 27th International Conference, ICONIP 2020, Bangkok, Thailand, November 23-27, 2020, Proceedings, Part II*, volume 12533 of *Lecture Notes in Computer Science*, pp. 189–201. Springer, 2020. doi: 10.1007/978-3-030-63833-7_16. URL https://doi.org/10.1007/978-3-030-63833-7_16.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32201–32223. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/liu24bs.html.

Fei Liu, Yiming Yao, Ping Guo, Zhiyuan Yang, Xi Lin, Zhe Zhao, Xialiang Tong, Kun Mao, Zhichao Lu, Zhenkun Wang, Mingxuan Yuan, and Qingfu Zhang. A systematic survey on large language models for algorithm design. *ACM Comput. Surv.*, 58(8):218:1–218:32, 2026. doi: 10.1145/3787585. URL https://doi.org/10.1145/3787585.

Jiawei Liu, Cheng Yang, Zhiyuan Lu, Junze Chen, Yibo Li, Mengmei Zhang, Ting Bai, Yuan Fang, Lichao Sun, Philip S. Yu, and Chuan Shi. Graph foundation models: Concepts, opportunities and challenges. *IEEE Trans. Pattern Anal. Mach. Intell.*, 47(6):5023–5044, 2025. doi: 10.1109/TPAMI.2025.3548729. URL https://doi.org/10.1109/TPAMI.2025.3548729.

Yang Liu and Jing Liu. A surrogate evolutionary neural architecture search algorithm for graph neural networks. *Appl. Soft Comput.*, 144:110485, 2023. doi: 10.1016/J.ASOC.2023.110485. URL https://doi.org/10.1016/j.asoc.2023.110485.

Shibing Mo, Kai Wu, Qixuan Gao, Xiangyi Teng, and Jing Liu. Autosgnn: Automatic propagation mechanism discovery for spectral graph neural networks. In Toby Walsh, Julie Shah, and Zico Kolter (eds.), *Thirty-Ninth AAAI Conference on Artificial Intelligence, Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence, Fifteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2025, Philadelphia, PA, USA, February 25 - March 4, 2025*, pp. 19493–19502. AAAI Press, 2025. doi: 10.1609/AAAI.V39I18.34146. URL https://doi.org/10.1609/aaai.v39i18.34146.

Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian,

M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery. *CoRR*, abs/2506.13131, 2025. doi: 10.48550/ARXIV.2506.13131. URL https://doi.org/10.48550/arXiv.2506.13131.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nat.*, 625(7995):468–475, 2024. doi: 10.1038/S41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

Min Shi, Yufei Tang, Xingquan Zhu, Yu Huang, David A. Wilson, Yuan Zhuang, and Jianxun Liu. Genetic-gnn: Evolutionary architecture search for graph neural networks. *Knowl. Based Syst.*, 247:108752, 2022. doi: 10.1016/J.KNOSYS.2022.108752. URL https://doi.org/10.1016/j.knosys.2022.108752.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K. Reddy. LLM-SR: scientific equation discovery via programming with large language models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=m2nmp8P5in.

Danny C. Sorensen. Implicit application of polynomial filters in a k-step arnoldi method. *SIAM J. Matrix Anal. Appl.*, 13(1):357–385, 1992. doi: 10.1137/0613025. URL https://doi.org/10.1137/0613025.

Petar Velickovic. Everything is connected: Graph neural networks. *CoRR*, abs/2301.08210, 2023. doi: 10.48550/ARXIV.2301.08210. URL https://doi.org/10.48550/arXiv.2301.08210.

Chao Wang, Jiaxuan Zhao, Lingling Li, Licheng Jiao, Fang Liu, Xu Liu, and Shuyuan Yang. Knowledge-aware evolutionary graph neural architecture search. *Knowl. Based Syst.*, 309: 112810, 2025a. doi: 10.1016/J.KNOSYS.2024.112810. URL https://doi.org/10.1016/j.knosys.2024.112810.

Haishuai Wang, Yang Gao, Xin Zheng, Peng Zhang, Hongyang Chen, and Jiajun Bu. Graph neural architecture search with GPT-4. *CoRR*, abs/2310.01436, 2023. doi: 10.48550/ARXIV.2310.01436. URL https://doi.org/10.48550/arXiv.2310.01436.

Haishuai Wang, Yang Gao, Xin Zheng, Peng Zhang, Jiajun Bu, and Philip S. Yu. Graph neural architecture search with large language models. *Sci. China Inf. Sci.*, 68(12), 2025b. doi: 10.1007/S11432-024-4539-1. URL https://doi.org/10.1007/s11432-024-4539-1.

Xiyuan Wang and Muhan Zhang. How powerful are spectral graph neural networks. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato (eds.), *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pp. 23341–23362. PMLR, 2022. URL https://proceedings.mlr.press/v162/wang22am.html.

Yi Wang, Jitao Zhao, Dongxiao He, Jia Li, Yuxiao Huang, and Zhiyong Feng. Topology-aware feature sorting enables universal modeling on homophilic and heterophilic graphs. In Hakim Hacid, Yoelle Maarek, Francesco Bonchi, Ido Guy, and Emine Yilmaz (eds.), *Proceedings of the ACM Web Conference 2026, WWW 2026, Dubai, United Arab Emirates, originally scheduled for April 13-17, 2026, rescheduled for June 29 - July 3, 2026*, pp. 475–486. ACM, 2026. doi: 10.1145/3774904.3792091. URL https://doi.org/10.1145/3774904.3792091.

Lanning Wei, Huan Zhao, and Zhiqiang He. Designing the topology of graph neural networks: A novel feature fusion perspective. In Frédérique Laforest, Raphaël Troncy, Elena Simperl, Deepak Agarwal, Aristides Gionis, Ivan Herman, and Lionel Médini (eds.), *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*, pp. 1381–1391. ACM, 2022. doi: 10.1145/3485447.3512185. URL https://doi.org/10.1145/3485447.3512185.

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Trans. Neural Networks Learn. Syst.*, 32 (1):4–24, 2021. doi: 10.1109/TNNLS.2020.2978386. URL https://doi.org/10.1109/TNNLS.2020.2978386.

Haobo Xu, Yuchen Yan, Dingsu Wang, Zhe Xu, Zhichen Zeng, Tarek F. Abdelzaher, Jiawei Han, and Hanghang Tong. SLOG: an inductive spectral graph neural network beyond polynomial filter. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 55348–55370. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/xu24aa.html.

Juwei Yue, Haikuo Li, Jiawei Sheng, Xiaodong Li, Taoyu Su, Tingwen Liu, and Li Guo. Hyperbolic-pde GNN: spectral graph neural networks in the perspective of A system of hyperbolic partial differential equations. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/yue25b.html.

Huan Zhao, Quanming Yao, and Weiwei Tu. Search to aggregate neighborhood for graph neural network. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pp. 552–563. IEEE, 2021. doi: 10.1109/ICDE51399.2021.00054. URL https://doi.org/10.1109/ICDE51399.2021.00054.

Shuai Zheng, Zhenfeng Zhu, Zhizhe Liu, Youru Li, and Yao Zhao. Node-oriented spectral filtering for graph neural networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 46(1):388–402, 2024. doi: 10.1109/TPAMI.2023.3324937. URL https://doi.org/10.1109/TPAMI.2023.3324937.

Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020. doi: 10.1016/J.AIOPEN.2021.01.001. URL https://doi.org/10.1016/j.aiopen.2021.01.001.

APPENDIX

LLM Usage: In this work, we used generative AI tools to assist with writing and polishing the manuscript, including language refinement, sentence rephrasing, grammar checking, readability improvement, and enhancing the overall flow and clarity of the text. We have not used generative AI tools for research ideation, research methodology, experimental design, data analysis, or other aspects of the scientific content, and these uses are not applicable to this work. Additionally, we used generative AI tools for language editing and proofreading. We have reviewed all AI-assisted work and manually verified the accuracy, consistency, and appropriateness of the resulting text. We take responsibility for the final content of this work, including all text, claims, analyses, and other content produced or polished with the aid of generative AI.

A THE DETAILS OF RELATED WORK

Spectral Graph Neural Networks. The core of spectral graph neural networks (SGNNs) lies in designing efficient spectral filters in the spectral domain to capture the key features of graph data. ChebNet (Defferrard, Bresson, and Vandergheynst 2016) and GCN (Kipf and Welling 2017) approximate the filters by Chebyshev polynomials, laying the foundation for SGNNs; GPRGNN (Chien et al. 2021) constructs adaptive filters by learning polynomial coefficients through end-to-end tasks; FAGCN (Bo et al. 2021) achieves adaptive filtering by approximating the weights of low-pass and high-pass filters using attention coefficients; BernNet (He et al. 2021) approximates filters by k-th-order Bernstein polynomials, enabling stable and efficient filter learning; Slog (Xu et al. 2024)

proposes a novel substantive filter that combines subgraph sampling in the spatial domain with signal processing in the spectral domain, thereby optimising inductive learning on large-scale graphs; NFGNN (Zheng et al. 2024) balances local and global spectral information to enhance both the model's expressive power and computational efficiency; Hyperbolic-PDE GNN (Yue et al. 2025) links message passing with SGNNs and enhances topological representation learning through physical information spectral filtering.

Graph Neural Architecture Search. GNAS aims to automate model architecture and hyperparameter selection. GraphNAS (Gao et al. 2019) and AutoGraph (Li and King 2020) search for the optimal architecture, respectively by reinforcement learning and genetic algorithms, within a predefined space; SANE (Zhao, Yao, and Tu 2021) employs a gradient-based, differentiable search strategy to reduce computational costs; F2GNN (Wei, Zhao, and He 2022) utilises a feature fusion framework to adaptively learn feature fusion weights for node pairs under different semantic contexts; Genetic-GNN (Shi et al. 2022) explores the search space of GNNs by designing novel crossover and mutation operators; CTFGNAS (Liu and Liu 2023) further extends the search space from network layers to topological connections and feature fusion strategies; KEGNAS (Wang et al. 2025) introduces a knowledge base to assist in the generation of graph learning architectures. However, GNAS is essentially a combinatorial optimisation problem constrained by expert-defined operators.

Graph Foundation Models. GFMs are pre-trained on large volumes of graph data to adapt to various downstream graph tasks; however, there is currently no standardised implementation approach. Depending on their relative reliance on GNNs and LLMs, existing methods fall into three categories (Liu et al. 2025): GNN-based foundation models (Gilmer et al. 2017; Ying et al. 2021; Wang et al. 2026), LLM-based foundation models (Wang et al. 2023; Ye et al. 2024), and foundation models that combine GNNs and LLMs (Yang et al. 2021; Mavromatis et al. 2023; Zhang et al. 2024).

B THE DETAILS OF BENCHMARK DATASETS

Table 7: The details of benchmark datasets.

Dataset	Classes	Nodes	Edges	Features
Cora	7	2,708	5,278	1,433
CiteSeer	6	3,327	4,552	3,703
PubMed	5	19,717	44,324	500
Computers	10	13,752	245,861	767
Photo	8	7,650	119,081	745
Chameleon	5	2,277	31,371	2,325
Squirrel	5	5,201	198,353	2,089
Texas	5	183	279	1,703
Cornell	5	183	277	1,703

To evaluate AgenticGNN on diverse graph structures, we select nine widely used benchmark datasets, detailed in Table 7. These datasets are categorized into two groups according to their graph feature homogeneity to assess the model's robustness across different graph characteristics. Homogeneous graphs included the citation networks Cora, CiteSeer and PubMed, and the Amazon co-purchase networks Computers and Photo. Heterogeneous graphs included the Wikipedia networks Chameleon and Squirrel, and the web networks Texas and Cornell.

C THE DETAILS OF GRAPH FEATURE PERCEPTION

Preliminary Investigation. To preliminary investigate the capability of proposed feature metrics in reflecting graph data characteristics, we visualize the graph data features metrics using radar charts (Fig 5). The charts reveal that graph datasets within the same domain exhibit highly similar structural profiles, while cross-domain datasets exhibit significant divergence. These domain-common patterns in graph features may provide transferable prior guidance for the design of graph learning algorithms by LLMs in automated graph learning.

Time Complexity. we provide a time complexity analysis for multi-dimensional graph feature perception. The total time complexity scales linearly with the graph size, bounded by

$\mathcal{O}(M \cdot F + K \cdot N + k \cdot M)$, proving an operational efficiency of AgenticGNN on large-scale graph. Let $N = |V|$ denote the total number of nodes, $M = |E|$ denote the number of edges, F denote the dimension of input node features, K denote the number of node categories, and d_{max} denote the maximum node degree. **Dirichlet Energy (ϵ_{dir}):** Calculating the feature differences requires traversing M edges. For feature vectors of dimension F, tensor subtraction and norm computation take $\mathcal{O}(F)$ per edge. Thus, the overall time complexity is $\mathcal{O}(M \cdot F)$. **Average Feature Cosine Similarity (S_{feat}):** Pre-normalizing node features across F dimensions takes $\mathcal{O}(N \cdot F)$. Traversing all edges to calculate the element-wise inner product takes $\mathcal{O}(M \cdot F)$. As $M \geq N - 1$ in connected graphs, the total complexity is $\mathcal{O}(M \cdot F)$. **Adjusted Edge Homogeneity Ratio (H_{adjusted}):** Calculating H_{edge} requires checking label equality over training edges in $\mathcal{O}(M)$ time. Computing the node degrees takes $\mathcal{O}(M)$ via `torch_geometric.utils.degree`. Summing class-wise degrees across K classes takes $\mathcal{O}(K \cdot N)$. Thus, the time complexity is $\mathcal{O}(M + K \cdot N)$. **Intra- and Inter-Class Path Lengths ($L_{\text{intra}}, L_{\text{inter}}$):** To avoid the expensive $\mathcal{O}(N(N + M))$ full-graph BFS, we samples a budget of S_{src} source nodes ($S_{src} \leq 200$) and executes a bounded BFS with a fixed depth cutoff $c = 5$. Let N_{local} and M_{local} be the average nodes and edges inside the c-hop neighborhood. The computation takes $\mathcal{O}(S_{src} \cdot (N_{local} + M_{local})) \ll \mathcal{O}(N(N + M))$, achieving scalable linear runtime relative to local density. **Degree Distribution Entropy (H_{deg}):** Node degrees are calculated in $\mathcal{O}(M)$ time, and degree frequency counting via `torch.bincount` takes $\mathcal{O}(N)$. Calculating log-probabilities takes $\mathcal{O}(d_{max})$. The total time complexity is $\mathcal{O}(M + N + d_{max}) = \mathcal{O}(M + N)$. **Spectral Radius ($\rho$) and Spectral Gap ($\Delta\lambda$):** Using Power Iteration and Hotelling Deflation, each iteration performs Sparse Matrix-Vector Multiplication (SpMV) using PyTorch's `index_add_`, taking $\mathcal{O}(M)$ time. For k iterations ($k \ll N$, set to 20 by default), the overall complexity is strictly bounded by $\mathcal{O}(k \cdot M)$.

D THE DETAILS OF SPECTRAL CHANGE FEEDBACK

Case I: Breakthrough on Chameleon. As plotted in Figure 6(a), the initial input spectrum of `Chameleon` exhibits a sharp energy concentration, with over 78% of its total energy aggregated within the 5-th and 6-th frequency bands, while its low-frequency energy is negligible. This spectral signature is highly consistent with the spatial quantization descriptors. The elevated Dirichlet energy ($\epsilon_{\text{dir}} = 5.0586$) and the diminished average feature similarity ($S_{\text{feat}} = 0.0159$) inherently reveals that the node features of `Chameleon` possess strong spatial heterogeneity and non-continuity across the graph topology. Consequently, conventional isotropic message passing, which acts as a low-pass filter, may over-smooth these highly informative mid-to-high frequency graph signals.The top-1 architecture generated by AgenticGNN performs global spectral energy debiasing and flattening. It reduces excessively high mid-frequency energy and transfers energy to lower and higher frequency bands. This explicit spectral energy redistribution improves the spatial representation discriminative power of node embeddings and achieves a test accuracy of 43.62% on a sparse graph of 2.5:2.5:95.

Case II: Limitation on PubMed. As plotted in Figure 6(b), LLM eliminate the minor energy residual in bands 5 and 6 using complex sinusoidal or logarithmic scaling. However, `PubMed` embodies a highly smooth topological structure, featuring a minuscule Dirichlet Energy ($\epsilon_{\text{dir}} = 0.1311$), high Edge Homophily ($H_{\text{edge}} = 0.7154$), and an extremely narrow Spectral Gap ($\Delta\lambda = 0.0078$). Due to the marginal spectral gap, these mid-to-high frequency components natively consist of structural noise rather than discriminative semantics. This hyper-micro-tuning in spectral domain performance bad in homophilic graph while it disrupts the global low-frequency smoothness, leading to stagnation or even negative optimization. Furthermore, the narrow spectral gap exerts a powerful topological hegemony during feature propagation, forcing the output spectrums of completely distinct structures to collapse into nearly identical low-pass shapes. This demonstrates that pure spectral feedback may fail to lead effective evolutionary on homophilic graph.

Case Summary. These comparative dissections firmly unveil that spectral fitting approach is inherently bounded. Pure spectral feedback serves as an invaluable compass for decoding heterophilic congestion, yet it may lose its efficacy or induce negative performance drifts under homophilic constraints. To overcome this dilemma, AgenticGNN abandons the blind reliance on singular spectral feedback. Instead, it prompts LLMs to diagnose the graph feature perception result and dynamically shifting toward spatial-domain structural augmentations when facing homophilic smooth configurations.

Time Complexity. Let $N = |V|$ denote the number of nodes, $M = |E|$ denote the number of

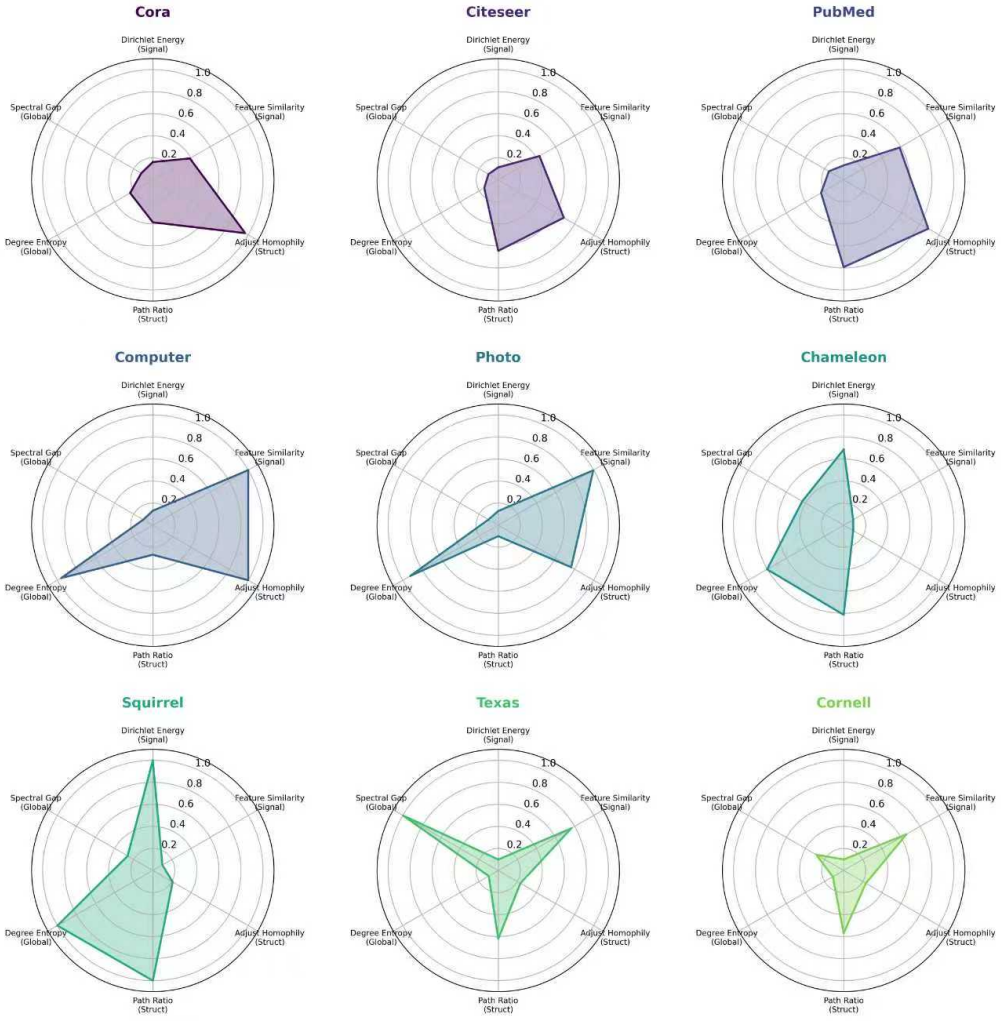
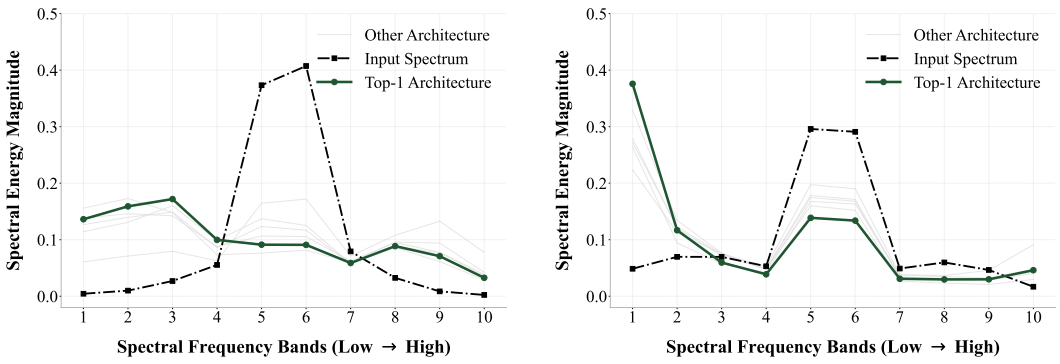


Figure 5: Radar Chart of Graph Feature.



(a) Chameleon (b) PubMed

Figure 6: Spectral change analysis on chameleon and pubmed datasets.

edges, d denote the feature dimension of the input node, K denote the order of Chebyshev polynomial approximation, B denote the number of Gaussian sub-bands, and P denote the resolution of numerical integration sampling points for pre-computing Chebyshev coefficients. The time com-

plexity scales strictly linearly with respect to M and d. Since $K \ll N$ and $B \ll N$ (typically $K = 30, B = 10$), this module achieves near-instantaneous spectral feedback in large-scale automated graph algorithm generation. **Filter Bank Initialization:** Constructing the normalized shifted Laplacian matrix $\tilde{L} = L - I_n$ requires extracting edge weights and subtracting unity on diagonal entries, which takes $\mathcal{O}(N + M)$ time. For each of the B Gaussian frequency sub-bands, computing the K-th order Chebyshev expansion coefficients $c_{m,k}$ involves numerical integration over P sampling points using numerical quadrature. Evaluating the cosine basis takes $\mathcal{O}(P \cdot K)$ per sub-band. Thus, the total offline initialization time is $\mathcal{O}(N + M + B \cdot P \cdot K)$. Since this phase is computed only once prior to model evaluation, it incurs zero overhead during algorithm iterations. **Chebyshev Basis Recursion:** To evaluate $g_m(\tilde{L})X$, rather than independently executing polynomial recursions for each sub-band, our module computes the shared K-th order Chebyshev basis tensor sequence $\{T_k(\tilde{L})X\}_{k=0}^{K}$ once. Calculating $T_1(\tilde{L})X = \tilde{L}X$ and the subsequent recurrence step $T_k(\tilde{L})X = 2\tilde{L}T_{k-1}(\tilde{L})X - T_{k-2}(\tilde{L})X$ requires a Sparse Matrix-Matrix Multiplication (SpMM) between $\tilde{L} \in \mathbf{R}^{N\times N}$ and $X \in \mathbf{R}^{N\times d}$. Given that $\tilde{L}$ has $\mathcal{O}(N + M)$ non-zero entries, each SpMM operation takes $\mathcal{O}(M \cdot d)$ time. Computing the entire sequence up to order K requires K matrix multiplications, yielding a time complexity of $\mathcal{O}(K \cdot M \cdot d)$. **Spectral Energy Density Filtering:** Given the cached Chebyshev bases $\{T_k(\tilde{L})X\}_{k=0}^{K}$, the filtered graph signal $Y_m = g_m(\tilde{L})X = \sum_{k=0}^{K} c_{m,k}T_k(\tilde{L})X$ for sub-band m is obtained via a weighted linear combination of tensors of size $N \times d$. Accumulating across K terms takes $\mathcal{O}(K \cdot N \cdot d)$ time. Computing the Frobenius norm energy $\|Y_m\|_F^2 = \mathrm{Tr}(Y_m^T Y_m)$ takes $\mathcal{O}(N \cdot d)$. Aggregating across all B frequency sub-bands and performing softmax energy normalization takes $\mathcal{O}(B \cdot K \cdot N \cdot d)$ time. **State Anomaly Feedback:** Checking for numerical divergence (NaN, $\pm\infty$, or norm overflow $\|T_k(\tilde{L})X\|_F > \tau$) is naturally integrated into the Chebyshev recursion loop, taking $\mathcal{O}(N \cdot d)$ per step with negligible constant-factor overhead.

E THE DETAILS OF CROSS-TASK KNOWLEDGE BASE

Construction of Normalized Graph Feature Vectors. To construct a robust and highly generalizable representation for our unified knowledge base, we design a 9-dimensional standardized graph feature vector. Each raw topological or spectral indicator is mapped into the range $[0, 1]$ using carefully engineered algebraic normalization functions. This ensures that the feature space is mathematically bounded and scale-invariant across different domains. The normalized vector $\mathbf{v} = [v_1, v_2, \ldots, v_9]^T \in [0, 1]^9$ is formulated as follows:

Dirichlet Energy (v_1): Measures node feature smoothness over the graph topology using an algebraic scaling function to handle unbounded values:

$$v_1 = \frac{E_{\text{Dir}}}{E_{\text{Dir}} + 1.0} \tag{16}$$

Feature Similarity (v_2): Normalizes the average cosine similarity of features between adjacent nodes from $[-1, 1]$ to $[0, 1]$:

$$v_2 = \frac{S_{\text{feat}} + 1.0}{2.0} \tag{17}$$

Adjusted Homophily (v_3): Scales the class-adjusted homophily index (which accounts for class imbalance and can be negative) linearly into the $[0, 1]$ range:

$$v_3 = \frac{H_{\text{adj}} + 1.0}{2.0} \tag{18}$$

Edge Homophily (v_4): Directly preserves the fraction of intra-class edges, which is naturally bounded:

$$v_4 = H_{\text{edge}} \tag{19}$$

Average Intra-class Path Length (v_5): Uses a nonlinear sigmoid-like scaling centered around a typical average path length of 3.0 to ensure high sensitivity:

$$v_5 = \frac{L_{\text{intra}}}{L_{\text{intra}} + 3.0} \tag{20}$$

Average Inter-class Path Length (v_6): Standardizes the geodesic distances between nodes of different classes with the same scale-sensitive scaling:

$$v_6 = \frac{L_{\text{inter}}}{L_{\text{inter}} + 3.0} \tag{21}$$

Path Ratio (v_7): Captures the relative structural separation between classes, which naturally resides within $[0, 1]$:

$$v_7 = R_{\text{path}} = \frac{L_{\text{intra}}}{L_{\text{inter}}} \tag{22}$$

Degree Distribution Entropy (v_8): Compresses the Shannon entropy of node degrees to quantify structural heterogeneity:

$$v_8 = \frac{H_{\text{deg}}}{H_{\text{deg}} + 3.0} \tag{23}$$

Spectral Radius (v_9): Scales the largest eigenvalue of the transition matrix (or adjacency-based operator) based on strict theoretical boundaries:

$$v_9 = \begin{cases} \frac{\rho}{2.0} & \text{if } \rho \leq 2.0 \\ 1.0 & \text{if } \rho > 2.0 \end{cases} \tag{24}$$

Role in Storage and Retrieval. By mapping diverse graph metrics to a unified and bounded space $[0, 1]^9$, this vector $\mathbf{v}$ serves as the coordinate system for our design knowledge base. This standardization guarantees that distance metrics, such as cosine similarity, can be calculated consistently, providing a rigorous mathematical foundation for zero-shot algorithm transfer and retrieval. We calculate the pairwise cosine similarity of standardized graph feature vectors across the 9 benchmark datasets (shown in Table 8) and group them into three distinct clusters identified by the PCA: (1) Cluster 1 including *Cora, Citeseer, Pubmed, Computers, Photo*; (2) Cluster 2 including *Chameleon* and *Squirrel*; (3) Cluster 3, including *Texas* and *Cornell*. The intra-cluster similarity for all three groups is remarkably high, the inter-cluster similarities are significantly lower. It ensures that target graph is accurately classified and matched with the most structurally compatible algorithmic designs in knowledge base, enabling reliable and robust zero-shot knowledge reuse.

Implementation Details of Knowledge Base Ablation. To rigorously guarantee the fairness, objectivity, and validity of our ablation studies regarding the global cross-task Knowledge Base (KB) and the zero-shot cold-start setting , we enforce strict data isolation and methodological controls throughout the experimental pipeline. **Iinitialization of the cross-task knowledge base:** The global cross-task Knowledge Base is constructed by aggregating effective evolutionary strategies and optimization experiences extracted from 9 distinct benchmark datasets. Since the construction of the initial knowledge base in the ablation study involves an independent and complete iterative process for each dataset, the specific execution order of the nine datasets during this phase has no impact on the final knowledge repository or subsequent ablation outcomes. **Prevention of Information leakage:** To prevent data contamination and ensure a fair evaluation of cross-task knowledge transfer, we implement a strict leave-one-out isolation strategy during the initialization phase: When evaluating a specific target dataset, the initialization phase systematically masks and excludes all historical experience, rules, and evolutionary trajectories associated exclusively with that target dataset from the knowledge base. By completely preventing any prior-experience leakage from the test dataset itself, we ensure that the performance gains observed in AgenticGNN stem strictly from genuine cross-task knowledge transfer rather than memorization or data contamination, thereby validating the rigor and fairness of the ablation study.

F Algorithm Generated by AgenticGNN

The following are the algorithms generated by AgenticGNN for node classification tasks across nine datasets, using a split ratio of 2.5%:2.5%:95%. Each algorithm includes its design reasoning, algorithm description, and code generated by large language model.

Table 8: Cross-Task Graph Topological Cosine Similarity.

Graph 1	Graph 2	Similarity
COMPUTERS	PHOTO	0.9989
CHAMELEON	SQUIRREL	0.9985
CITESEER	PUBMED	0.9975
TEXAS	CORNELL	0.9974
CORA	PUBMED	0.9969
CITESEER	CORA	0.9932
PUBMED	COMPUTERS	0.9912
CORA	PHOTO	0.9911
CORA	COMPUTERS	0.9904
PUBMED	PHOTO	0.9895
CITESEER	COMPUTERS	0.9891
CITESEER	PHOTO	0.9865
CITESEER	CORNELL	0.9544
PUBMED	CORNELL	0.9382
CORNELL	COMPUTERS	0.9373
CITESEER	TEXAS	0.9315
CORNELL	PHOTO	0.9249
CORA	CORNELL	0.9194
TEXAS	COMPUTERS	0.9142
PUBMED	TEXAS	0.9127
CHAMELEON	CORNELL	0.9039
TEXAS	PHOTO	0.8999
CHAMELEON	TEXAS	0.8984
CORA	TEXAS	0.8912
SQUIRREL	CORNELL	0.8840
CHAMELEON	COMPUTERS	0.8823
CITESEER	CHAMELEON	0.8820
SQUIRREL	TEXAS	0.8798
CORA	CHAMELEON	0.8796
PUBMED	CHAMELEON	0.8788
CHAMELEON	PHOTO	0.8702
SQUIRREL	COMPUTERS	0.8601
CITESEER	SQUIRREL	0.8576
CORA	SQUIRREL	0.8564
PUBMED	SQUIRREL	0.8549
SQUIRREL	PHOTO	0.8470

F.1 CORA

Design Reasoning:
Considering the Cora graph's homophilic structure, it is crucial to prioritize the middle-frequency spectrum for embedding effectiveness while leveraging minimal parametric complexity to ensure robustness against overfitting.
Algorithm Description:
A novel spectral graph neural network layer is proposed that adapts propagation through direct self-weighted embeddings and dynamic middle-frequency enhancement via median filtering across multiple hop-aware propagation steps.
Code:

```
import torch
import torch_geometric.utils
from torch_geometric.nn.conv import MessagePassing

class GNN_Layer(MessagePassing):
    def __init__(self, input_channels, output_channels):
        super(GNN_Layer, self).__init__(aggr='add')
        self.K = 4  # Number of propagation steps
```

```
        self.middle_scaling = 0.7  # Middle-frequency scaling factor
        self.residual_weight = 0.3  # Residual self-weight

    def forward(self, x, x_raw, edge_index):
        num_nodes = x.shape[0]
        edge_index, _ = torch_geometric.utils.add_remaining_self_loops(
            edge_index, None, 1.0, num_nodes)
        edge_weight = torch.ones((edge_index.size(1),), dtype=x.dtype,
            device=edge_index.device)

        row, col = edge_index[0], edge_index[1]
        deg = torch_geometric.utils.scatter(edge_weight, col, dim=0,
            dim_size=num_nodes, reduce='sum')
        deg_inv_sqrt = deg.pow(-0.5)
        deg_inv_sqrt.masked_fill_(deg_inv_sqrt == float('inf'), 0)
        edge_weight = deg_inv_sqrt[row] * edge_weight * deg_inv_sqrt[
            col]

        # Initialize hidden representation with residual weighted raw
            features
        hidden = x_raw * self.residual_weight
        for k in range(self.K):
            x = self.propagate(edge_index, x=x, norm=edge_weight)

            # Apply dynamic middle-frequency enhancement using a median
                filtering approach
            median_factor = (1.0 / (1 + abs(k - (self.K // 2)))) * self
                .middle_scaling
            hidden += median_factor * x

        return hidden

    def message(self, x_j, norm):
        return norm.view(-1, 1) * x_j
```

F.2 CITESEER

Design Reasoning:
Citeseer graph exhibits semi-homophilic properties, necessitating balanced energy redistribution across low, medium, and high-frequency bands to ensure both global smoothness and local discriminatory feature learning.
Algorithm Description:
Introduce a spectral GNN layer with a dynamically learned polynomial transformation on normalized adjacency matrices using a single set of weights shared across scales to prevent overfitting and achieve uniform spectral filtration.
Code:

```
import torch
import numpy as np
import torch_geometric.utils
from torch.nn import Parameter
from torch_geometric.nn.conv import MessagePassing

class GNN_Layer(MessagePassing):
    def __init__(self, input_channels, output_channels):
        super(GNN_Layer, self).__init__(aggr='add')
        self.K = 5  # Number of propagation steps
        self.weights = Parameter(torch.tensor([0.5] * self.K))  #
            Shared weights for spectral filtering
```

```
        self.alpha = Parameter(torch.tensor(0.3))  # Parameter to blend
            raw and learned features
        self.reset_parameters()

    def reset_parameters(self):
        torch.nn.init.uniform_(self.weights, a=0.1, b=0.9)
        self.alpha.data = torch.tensor(0.3)

    def forward(self, x, x_raw, edge_index):
        num_nodes = x.shape[0]
        edge_index, _ = torch_geometric.utils.add_remaining_self_loops(
            edge_index, None, 1.0, num_nodes)
        edge_weight = torch.ones((edge_index.size(1),), dtype=x.dtype,
            device=edge_index.device)

        row, col = edge_index[0], edge_index[1]
        deg = torch_geometric.utils.scatter(edge_weight, col, dim=0,
            dim_size=num_nodes, reduce='sum')
        deg_inv_sqrt = deg.pow(-0.5)
        deg_inv_sqrt.masked_fill_(deg_inv_sqrt == float('inf'), 0)
        edge_weight = deg_inv_sqrt[row] * edge_weight * deg_inv_sqrt[
            col]

        hidden = x_raw * self.alpha  # Start with raw features weighted
            by \alpha
        for k in range(self.K):
            x = self.propagate(edge_index, x=x, norm=edge_weight)
            hidden += self.weights[k] * x  # Apply shared weights for
                all propagation steps

        hidden = hidden + (1 - self.alpha) * x_raw  # Blend propagated
            and raw features
        return hidden

    def message(self, x_j, norm):
        return norm.view(-1, 1) * x_j
```

F.3 PUBMED

Design Reasoning:
A novel GNN layer architecture that combines graph attention mechanisms with learnable edge representations and adaptive feature transformations for improved node representation learning in citation network graphs.
Algorithm Description:
A novel GNN layer architecture that combines graph attention mechanisms with learnable edge representations and adaptive feature transformations for improved node representation learning in citation network graphs.
Code:

```
import torch
import torch_geometric.utils
from torch.nn import Parameter
from torch_geometric.nn.conv import MessagePassing

class GNN_Layer(MessagePassing):
    def __init__(self, input_channels, output_channels):
        super(GNN_Layer, self).__init__(aggr='add')
        self.K = 5
        self.alpha = 0.4
        self.beta = Parameter(torch.tensor([0.3]*self.K))
```

```
        self.p = Parameter(torch.tensor(0.6))

    def reset_parameters(self):
        torch.nn.init.ones_(self.beta)
        self.p.data = 0.6

    def forward(self, x, x_raw, edge_index):
        num_nodes = x.shape[0]
        edge_index, _ = torch_geometric.utils.add_remaining_self_loops(
            edge_index, None, 2., num_nodes)
        edge_weight = torch.ones((edge_index.size(1),), dtype=x.dtype,
            device=edge_index.device)
        row, col = edge_index[0], edge_index[1]
        idx = col
        deg = torch_geometric.utils.scatter(edge_weight, idx, dim=0,
            dim_size=num_nodes, reduce='sum')
        deg_inv_sqrt = deg.pow_(-0.5)
        deg_inv_sqrt.masked_fill_(deg_inv_sqrt == float('inf'), 0)
        edge_weight = deg_inv_sqrt[row] * edge_weight * deg_inv_sqrt[
            col]

        hidden = x + self.alpha * x_raw
        for k in range(self.K):
            x = self.propagate(edge_index, x=x, norm=edge_weight)
            beta = torch.sigmoid(self.beta[k])
            attention_scores = torch.sigmoid(self.p)
            x = x * attention_scores
            hidden = hidden + beta * x

        return hidden

    def message(self, x_j, norm):
        return norm.view(-1, 1) * x_j
```

F.4 COMPUTERS

Design Reasoning:
The common backbone idea in the provided algorithms is to design adaptive spectral filters (e.g., cosine modulation, wavelet modulation, Laplacian regularization) that redistribute energy across frequency bands while leveraging residual connections and normalization for stable propagation. The target graph, exhibiting medium-frequency dominance, benefits mostly from boosting interactions around the medium-frequency spectrum without losing low-frequency smoothness or triggering a high-frequency noise overflow. Inspired by this, we propose a new design utilizing adaptive cubic polynomial filters to emphasize medium-frequency interactions. The cubic filters offer higher flexibility to fine-tune the energy around the medium spectrum, paired with energy scaling normalization, attention-guided propagation, and smooth residual initialization to ensure stable optimization.
Algorithm Description:
A cubic polynomial-modulated spectral graph neural network layer with dynamic energy scaling, adaptive attention propagation, and residual connections tailored for medium-frequency enhancement while stabilizing feature transitions.
Code:

```
import torch
import torch.nn.functional as F
from torch.nn import Parameter, Linear
from torch_geometric.nn.conv import MessagePassing
import torch_geometric.utils
```

```
class GNN_Layer(MessagePassing):
    def __init__(self, input_channels, output_channels):
        super(GNN_Layer, self).__init__(aggr='add')
        self.K = 3  # Degree of cubic polynomial
        self.linear = Linear(input_channels, output_channels, bias=
            False)
        self.polynomial_weights = Parameter(torch.ones(self.K))  #
            Adaptive cubic weights
        self.energy_scale = Parameter(torch.tensor(0.5))  # Energy
            scaling factor
        self.residual_weight = Parameter(torch.tensor(0.6))  # Residual
             connection weight
        self.attention = Parameter(torch.tensor(0.4))  # Attention
            weight
        self.reset_parameters()

    def reset_parameters(self):
        torch.nn.init.ones_(self.polynomial_weights)
        self.energy_scale.data.fill_(0.5)
        self.residual_weight.data.fill_(0.6)
        self.attention.data.fill_(0.4)
        torch.nn.init.xavier_uniform_(self.linear.weight)

    def forward(self, x, x_raw, edge_index):
        num_nodes = x.size(0)
        edge_index, _ = torch_geometric.utils.add_remaining_self_loops(
            edge_index, None, fill_value=1.0, num_nodes=num_nodes)
        edge_weight = torch.ones((edge_index.size(1),), dtype=x.dtype,
            device=x.device)
        row, col = edge_index[0], edge_index[1]
        deg = torch_geometric.utils.degree(col, num_nodes, dtype=x.
            dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        deg_inv_sqrt[deg_inv_sqrt == float('inf')] = 0
        edge_weight = deg_inv_sqrt[row] * edge_weight * deg_inv_sqrt[
            col]

        % Transform features using linear mapping
        transformed_x = self.linear(x)

        % Initialize hidden representation with residual connection
        hidden = x_raw * self.residual_weight

        % Apply cubic polynomial filtering with adaptive attention
        for k in range(self.K):
            polynomial_term = self.polynomial_weights[k] ** (k + 1)  %
                Cubic polynomial modulation
            propagated_x = self.propagate(edge_index, x=transformed_x,
                norm=edge_weight)
            attention_score = torch.sigmoid(self.attention)
            scaled_contribution = propagated_x * polynomial_term * self
                .energy_scale
            hidden += attention_score * scaled_contribution

        % L2 normalization for stability
        hidden = F.normalize(hidden, p=2, dim=1)

        return hidden

    def message(self, x_j, norm):
        return norm.view(-1, 1) * x_j
```

F.5 PHOTO

Design Reasoning:
Leveraging graphs with mixed feature distributions requires balancing local feature refinement and global structural aggregation while simultaneously ensuring energy preservation across mid-frequency bands, thus minimizing overfitting and enhancing generalization.
Algorithm Description:
Introduce a spectral graph layer combining localized Laplacian eigenvector filtering with global feature-channel pooling inspired by directed diffusion mechanisms, coupling these with matrix-variate low-rank projections for scalable, energy-focused embeddings.
Code:

```
import torch
import torch.nn.functional as F
from torch.nn import Parameter
from torch_geometric.nn.conv import MessagePassing
from torch_geometric.utils import add_remaining_self_loops, scatter

class GNN_Layer(MessagePassing):
    def __init__(self, input_channels, output_channels):
        super(GNN_Layer, self).__init__(aggr='add')  # Aggregation
            method
        self.K_local = 3  # Number of localized Laplacian filtering
            hops
        self.K_global = 3  # Steps for global channel pooling
        self.alpha = 0.6  # Feature blending coefficient
        self.laplacian_weights = Parameter(torch.ones(self.K_local))  #
             Learnable weights for Laplacian filtering
        self.proj_global = torch.nn.Linear(input_channels,
            output_channels, bias=False)  # Global feature projection
        self.low_rank_proj = torch.nn.Linear(input_channels,
            output_channels, bias=False)  # Low-rank matrix projection

    def reset_parameters(self):
        torch.nn.init.xavier_uniform_(self.proj_global.weight)  #
            Initialize projection weights
        torch.nn.init.xavier_uniform_(self.low_rank_proj.weight)  #
            Initialize low-rank weights
        torch.nn.init.constant_(self.laplacian_weights, 1.0)  #
            Initialize Laplacian weights

    def forward(self, x, x_raw, edge_index):
        num_nodes = x.size(0)
        edge_index, _ = add_remaining_self_loops(edge_index, None, 1.0,
             num_nodes)  # Add self-loops
        edge_weight = torch.ones((edge_index.size(1),), dtype=x.dtype,
            device=edge_index.device)
        row, col = edge_index[0], edge_index[1]
        deg = scatter(edge_weight, col, dim=0, dim_size=num_nodes,
            reduce='sum')
        deg_inv_sqrt = deg.pow(-0.5).masked_fill(deg == 0, 0)
        edge_weight = deg_inv_sqrt[row] * edge_weight * deg_inv_sqrt[
            col]

        # Local filtering using Laplacian eigenvector approximation
        local_signal = x
        local_hidden = x * 0
        for k in range(self.K_local):
            local_signal = self.propagate(edge_index, x=local_signal,
                norm=edge_weight)
            local_hidden += torch.sigmoid(self.laplacian_weights[k]) *
                local_signal
```

```
        # Global feature-channel pooling via directed diffusion
        global_signal = x_raw
        global_hidden = self.proj_global(global_signal)
        for k in range(self.K_global):
            global_signal = self.propagate(edge_index, x=global_signal,
                norm=edge_weight)
        global_hidden += global_signal

        # Combine local, global, and matrix-variate low-rank features
        combined_features = local_hidden + global_hidden
        hidden = self.alpha * self.low_rank_proj(x_raw) + (1 - self.
            alpha) * combined_features

        # Normalize final feature representation
        hidden = F.normalize(hidden, p=2, dim=-1)
        return hidden

    def message(self, x_j, norm):
        return norm.view(-1, 1) * x_j
```

F.6 CHAMELEON

Design Reasoning:
Across the provided spectral algorithms, the common backbone idea centers on spectral filtering to shift energy across frequency bands, with low-frequency enhancement and residual connections mitigating instabilities while adapting to heterophilic graphs like Chameleon. Observing dominant performance at algorithms leveraging balanced spectral energy across low, mid, and high frequencies, the new design adapts by introducing explicitly decoupled frequency channels modulated through signal-controlled band-pass filters, embedded within a hierarchical learning framework to stabilize frequency migration and align node embeddings for heterophily.
Algorithm Description:
Develop a spectral GNN layer using hierarchical frequency decomposition with channel-wise learnable modulation filters, adaptive normalization, and residual unification to capture the Chameleon graph's diverse spectrum dynamics.
Code:

```
import torch
import torch_geometric.utils
from torch.nn import Parameter
from torch_geometric.nn.conv import MessagePassing

class GNN_Layer(MessagePassing):
    def __init__(self, input_channels, output_channels):
        super(GNN_Layer, self).__init__(aggr='add')
        self.low_weight = Parameter(torch.randn(input_channels,
            output_channels))  # Low-frequency modulated filter
        self.mid_weight = Parameter(torch.randn(input_channels,
            output_channels))  # Mid-frequency modulated filter
        self.high_weight = Parameter(torch.randn(input_channels,
            output_channels))  # High-frequency modulated filter

        # Learnable modulation coefficients for frequency channels
        self.filter_low = Parameter(torch.tensor(0.4))
        self.filter_mid = Parameter(torch.tensor(0.3))
        self.filter_high = Parameter(torch.tensor(0.2))

        # Learnable scaling and residual controls
        self.normalization_scale = Parameter(torch.tensor(1.0))
        self.residual_scale = Parameter(torch.tensor(0.5))
```

```
    def reset_parameters(self):
        torch.nn.init.xavier_uniform_(self.low_weight)
        torch.nn.init.xavier_uniform_(self.mid_weight)
        torch.nn.init.xavier_uniform_(self.high_weight)
        torch.nn.init.constant_(self.filter_low, 0.4)
        torch.nn.init.constant_(self.filter_mid, 0.3)
        torch.nn.init.constant_(self.filter_high, 0.2)
        torch.nn.init.constant_(self.normalization_scale, 1.0)
        torch.nn.init.constant_(self.residual_scale, 0.5)

    def forward(self, x, x_raw, edge_index):
        num_nodes = x.shape[0]

        # Add self-loops and normalize edge weights
        edge_index, _ = torch_geometric.utils.add_remaining_self_loops(
            edge_index, None, 1.0, num_nodes)
        edge_weight = torch.ones(edge_index.size(1), dtype=x.dtype,
            device=edge_index.device)
        row, col = edge_index[0], edge_index[1]
        deg = torch_geometric.utils.scatter(edge_weight, col, dim=0,
            dim_size=num_nodes, reduce='sum')
        deg_inv_sqrt = deg.pow(-0.5).masked_fill(deg == 0, 0)
        edge_weight = deg_inv_sqrt[row] * edge_weight * deg_inv_sqrt[
            col]

        # Hierarchical frequency decomposition
        # Low-frequency channel
        low_pass = self.propagate(edge_index, x=x, norm=edge_weight)
        low_pass = torch.matmul(low_pass, self.low_weight) * self.
            filter_low

        # Mid-frequency channel
        mid_pass = x
        for _ in range(2):
            mid_pass = self.propagate(edge_index, x=mid_pass, norm=
                edge_weight)
        mid_pass = torch.matmul(mid_pass, self.mid_weight) * self.
            filter_mid

        # High-frequency channel
        high_pass = x_raw
        for _ in range(4):
            high_pass = self.propagate(edge_index, x=high_pass, norm=
                edge_weight)
        high_pass = torch.matmul(high_pass, self.high_weight) * self.
            filter_high

        # Unification with residual connections and adaptive
            normalization
        residual = x_raw * self.residual_scale
        hidden = low_pass + mid_pass + high_pass + residual
        hidden_norm = torch.norm(hidden, dim=1, keepdim=True).clamp(min
            =1e-9)
        hidden = hidden / hidden_norm * self.normalization_scale

        return hidden

    def message(self, x_j, norm):
        return norm.view(-1, 1) * x_j
```

F.7 SQUIRREL

Design Reasoning:
The Squirrel graph, being highly heterogeneous, benefits from multi-scale aggregations and attention mechanisms, motivating the design to incorporate hierarchical processing and self-attention for distinguishing diverse node roles effectively.
Algorithm Description:
The new algorithm uses a multi-scale hierarchical aggregation approach combined with self-attention mechanisms to enable better feature extraction and context-sensitive node embeddings.
Code:

```
import torch
import torch_geometric.utils
from torch.nn import Parameter, Linear
from torch.nn.functional import relu, softmax
from torch_geometric.nn.conv import MessagePassing

class GNN_Layer(MessagePassing):
    def __init__(self, input_channels, output_channels):
        super(GNN_Layer, self).__init__(aggr='add')
        self.scale_levels = 3  # Number of hierarchical aggregation
            levels
        self.attention_weights = Parameter(torch.Tensor(self.
            scale_levels))  # Trainable self-attention weights
        self.local_transforms = torch.nn.ModuleList(
            [Linear(input_channels, output_channels) for _ in range(
                self.scale_levels)]
        )  # Multi-scale transformations
        self.global_transform = Linear(input_channels, output_channels)
              # Global feature transformation
        self.combine_transform = Linear(2 * output_channels,
            output_channels)  # Combined feature transformation

        self.reset_parameters()

    def reset_parameters(self):
        torch.nn.init.uniform_(self.attention_weights, a=0, b=1)
        for transform in self.local_transforms:
            transform.reset_parameters()
        self.global_transform.reset_parameters()
        self.combine_transform.reset_parameters()

    def forward(self, x, x_raw, edge_index):
        num_nodes = x.shape[0]

        # Add self-loops and normalize edges
        edge_index, _ = torch_geometric.utils.add_remaining_self_loops(
            edge_index, fill_value=1.0, num_nodes=num_nodes)
        edge_weight = torch.ones((edge_index.size(1),), dtype=x.dtype,
            device=edge_index.device)
        row, col = edge_index[0], edge_index[1]
        deg = torch_geometric.utils.scatter(edge_weight, col, dim=0,
            dim_size=num_nodes, reduce='sum')
        deg_inv_sqrt = deg.pow(-0.5).clamp(min=0)
        edge_weight = deg_inv_sqrt[row] * edge_weight * deg_inv_sqrt[
            col]

        # Multi-scale hierarchical aggregation
        multi_scale_features = []
        scale_x = x
        for i in range(self.scale_levels):
            scale_x = self.propagate(edge_index, x=scale_x, norm=
                edge_weight)
```

```
            transformed = relu(self.local_transforms[i](scale_x))
            multi_scale_features.append(transformed)

        # Weight the features across scales using self-attention
        attention_scores = softmax(self.attention_weights, dim=0)
        hierarchical_hidden = torch.zeros_like(multi_scale_features[0])
        for i in range(self.scale_levels):
            hierarchical_hidden += attention_scores[i] *
                multi_scale_features[i]

        # Global feature transformation
        global_features = relu(self.global_transform(x_raw.mean(dim=0).
            view(1, -1).repeat(num_nodes, 1)))\n
        # Combine hierarchical and global features
        combined_features = torch.cat([hierarchical_hidden,
            global_features], dim=-1)
        hidden = self.combine_transform(combined_features)

        return hidden

    def message(self, x_j, norm):
        return norm.view(-1, 1) * x_j
```

F.8 TEXAS

Design Reasoning:
The proposed design leverages a Chebyshev polynomial-based spectral filtering mechanism for smooth frequency decomposition, which is suitable for the Texas graph's heterophilic structure due to its capability to adaptively balance high-frequency and low-frequency information.
Algorithm Description:
The network layer employs Chebyshev polynomial approximations with residual connections and a learnable attention mechanism, enabling effective node embeddings and improving node classification.
Code:

```
import torch
import torch_geometric.utils
from torch.nn import Parameter
from torch_geometric.nn.conv import MessagePassing

class GNN_Layer(MessagePassing):
    def __init__(self, input_channels, output_channels):
        super(GNN_Layer, self).__init__(aggr='add')
        self.input_channels = input_channels
        self.output_channels = output_channels
        self.K = 3  # Number of Chebyshev polynomial orders
        self.attention_weights = Parameter(torch.Tensor(self.K))  #
            Learnable attention weights for polynomial filters
        self.feature_projection = torch.nn.Linear(input_channels,
            output_channels)
        self.residual_weight = Parameter(torch.Tensor(input_channels,
            output_channels))
        self.merge_layer = torch.nn.Linear(output_channels,
            output_channels)
        self.reset_parameters()

    def reset_parameters(self):
        torch.nn.init.constant_(self.attention_weights, 1.0 / self.K)
        torch.nn.init.xavier_uniform_(self.residual_weight)
        self.feature_projection.reset_parameters()
```

```
        self.merge_layer.reset_parameters()

    def chebyshev_filter(self, x, edge_index, edge_weight, k):
        num_nodes = x.size(0)
        edge_index, edge_weight = torch_geometric.utils.get_laplacian(
            edge_index, normalization='sym', num_nodes=num_nodes)
        tx_prev = x
        tx_curr = self.propagate(edge_index, x=x, norm=edge_weight)
        for _ in range(1, k):
            tx_next = 2 * self.propagate(edge_index, x=tx_curr, norm=
                edge_weight) - tx_prev
            tx_prev, tx_curr = tx_curr, tx_next
        return tx_curr

    def forward(self, x, x_raw, edge_index):
        num_nodes = x.size(0)
        edge_index, _ = torch_geometric.utils.add_remaining_self_loops(
            edge_index, None, 1.0, num_nodes)
        edge_weight = torch.ones((edge_index.size(1),), dtype=x.dtype,
            device=x.device)
        row, col = edge_index[0], edge_index[1]

        deg = torch_geometric.utils.scatter(edge_weight, col, dim=0,
            dim_size=num_nodes, reduce='sum')
        deg_inv_sqrt = deg.pow(-0.5)
        deg_inv_sqrt.masked_fill_(deg_inv_sqrt == float('inf'), 0)
        edge_weight = deg_inv_sqrt[row] * edge_weight * deg_inv_sqrt[
            col]

        initial_features = self.feature_projection(x_raw)
        residual_features = x_raw @ self.residual_weight
        hidden = initial_features

        for k in range(1, self.K + 1):
            chebyshev_output = self.chebyshev_filter(hidden, edge_index
                , edge_weight, k)
            hidden = hidden + torch.sigmoid(self.attention_weights[k -
                1]) * chebyshev_output

        hidden = hidden + residual_features
        hidden = self.merge_layer(hidden)
        return hidden

    def message(self, x_j, norm):
        return norm.view(-1, 1) * x_j
```

F.9 CORNELL

Design Reasoning:
The common backbone in the provided algorithms is the utilization of spectral graph theory, Chebyshev polynomial approximations, and multi-scale feature aggregation. To adapt and improve for the target Cornell graph, we propose leveraging hybrid hierarchical filtering by combining both Chebyshev filtering and adaptive Fourier-based frequency band extraction. This integration helps in capturing both high-frequency and low-frequency node interactions, which is crucial for graphs with heterophily and modularity like the Cornell graph.
Algorithm Description:
A new GNN layer is introduced that merges Chebyshev polynomial spectral filters with adaptively learned Fourier band-pass filters, enabling more precise separation of neighborhoods in the feature space.
Code:

```
import torch
import torch_geometric.utils
from torch.nn import Parameter
from torch_geometric.nn.conv import MessagePassing
import torch.nn.functional as F

class GNN_Layer(MessagePassing):
    def __init__(self, input_channels, output_channels):
        super(GNN_Layer, self).__init__(aggr='add')
        self.K = 3  # Chebyshev polynomial order
        self.cheb_weights = Parameter(torch.rand(self.K, input_channels
            ))
        self.fourier_weights = Parameter(torch.rand(input_channels,
            input_channels))  # Fourier filter weights
        self.residual_transform = torch.nn.Linear(input_channels,
            output_channels)
        self.scale_transform = torch.nn.Linear(2 * input_channels,
            output_channels)
        self.reset_parameters()

    def reset_parameters(self):
        torch.nn.init.uniform_(self.cheb_weights, a=0.1, b=0.5)
        torch.nn.init.xavier_uniform_(self.fourier_weights)
        torch.nn.init.xavier_uniform_(self.residual_transform.weight)
        torch.nn.init.xavier_uniform_(self.scale_transform.weight)

    def forward(self, x, x_raw, edge_index):
        num_nodes = x.shape[0]
        edge_index, _ = torch_geometric.utils.add_remaining_self_loops(
            edge_index, None, fill_value=1.0, num_nodes=num_nodes)
        edge_weight = torch.ones((edge_index.size(1),), device=x.device
            , dtype=x.dtype)

        # Degree normalization
        row, col = edge_index[0], edge_index[1]
        deg = torch_geometric.utils.degree(row, num_nodes, dtype=x.
            dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        deg_inv_sqrt.masked_fill_(deg_inv_sqrt == float('inf'), 0)
        edge_weight = deg_inv_sqrt[row] * edge_weight * deg_inv_sqrt[
            col]

        # Chebyshev polynomial spectral filtering
        cheb_features = [x]
        Tx_prev, Tx_curr = x, self.propagate(edge_index, x=x, norm=
            edge_weight)

        # Generate higher-order Chebyshev terms
        for k in range(2, self.K):
            Tx_next = 2 * self.propagate(edge_index, x=Tx_curr, norm=
                edge_weight) - Tx_prev
            Tx_prev, Tx_curr = Tx_curr, Tx_next
            cheb_features.append(self.cheb_weights[k - 1] * Tx_next)

        # Combine filtered results
        cheb_output = sum(cheb_features)

        # Fourier-based band-pass filtering
        fourier_output = torch.mm(x, self.fourier_weights)

        # Hybrid Feature Combination
        hybrid_features = cheb_output + fourier_output  # Fusion of
            spectral contributions
```

```
        # Residual connection
        residual = self.residual_transform(x_raw)

        # Final feature concatenation
        concatenated_features = torch.cat([hybrid_features, residual],
            dim=-1)
        hidden = self.scale_transform(concatenated_features)

        return hidden

    def message(self, x_j, norm):
        return norm.view(-1, 1) * x_j
```

G Analyze of Core Parameter

Population Size. To evaluate the robustness and stability of AgenticGNN, we conduct parameter sensitivity analyses on the evolutionary population size and the LLM generation core. We test the influence of the evolutionary population size $P \in \{10, 20, 30, 40, 50\}$ on representative datasets. As shown in Table 9, a moderate population size of 30 strikes a better balance between search performance and computational overhead.

Core LLM Model. Since our main framework adopts GPT-4o as the generation core, we further investigate different foundational LLMs under the same experiment setup and prompt templates. Results are shown in Table 10.

Table 9: Sensitivity study of evolutionary population size P in terms of Test Accuracy (%).

Dataset	$P = 10$	$P = 20$	$P = 30$	$P = 40$	$P = 50$
Pubmed	$\mathbf{85.75 \pm 0.29}$	85.28 ± 0.38	85.17 ± 0.24	$\underline{85.50 \pm 0.41}$	85.27 ± 0.51
Photo	91.61 ± 0.88	$\underline{92.11 \pm 0.42}$	$\mathbf{92.50 \pm 0.28}$	91.91 ± 0.59	92.00 ± 1.02
Chameleon	$\underline{40.24 \pm 1.99}$	39.74 ± 3.38	$\mathbf{43.62 \pm 3.43}$	39.75 ± 3.38	39.76 ± 3.39
Texas	$\mathbf{64.05 \pm 3.69}$	62.31 ± 11.15	62.83 ± 3.12	61.97 ± 7.42	$\underline{63.01 \pm 1.88}$
Cornell	51.91 ± 9.28	52.89 ± 7.60	$\underline{53.99 \pm 9.05}$	52.20 ± 12.09	$\mathbf{54.80 \pm 4.39}$

Table 10: Comparison using different LLMs as the generation core in terms of Test Accuracy (%).

Dataset	gpt-4o-mini	gpt-4o	gpt-5.2
Texas	52.54 ± 6.86	$\mathbf{62.83 \pm 3.12}$	$\underline{61.45 \pm 6.55}$

H Visualization of Experiental Results

In this section, we present additional empirical visualization of experimental findings in the main paper. Figure 7 illustrates the ablation curve on the Squirrel dataset. Figure 8 presents the PCA projection and radar chart of normalized graph features. Figure 9 illustrates the heatmap of node-classification ranks across different training-test splits, demonstrating the performance consistency and stability of AgenticGNN under varying data regimes. Finally, Figure 10 displays the accuracy curves across all nine benchmark datasets in the knowledge-base ablation study.

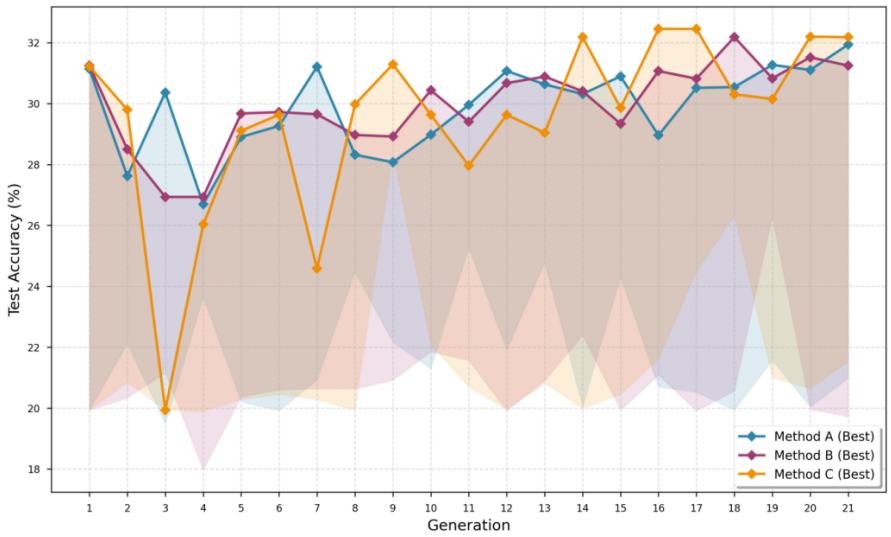
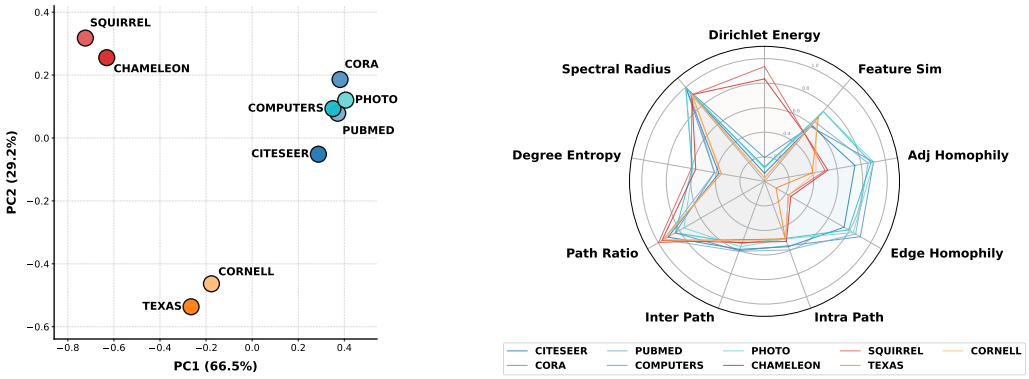


Figure 7: Ablation curve on squirrel node-classification task.



(a) PCA projection

(b) Radar chart

Figure 8: PCA projection and radar chart of normalized graph feature vectors.

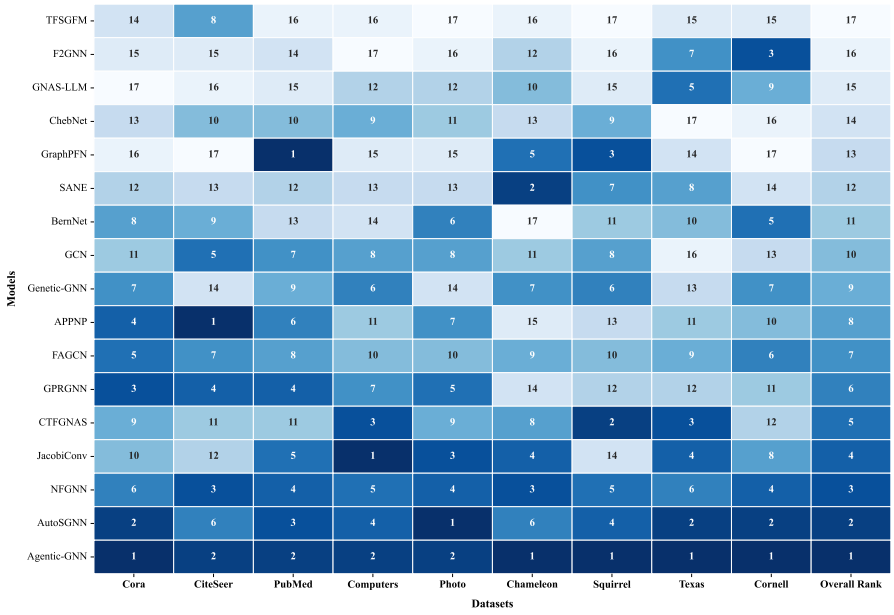
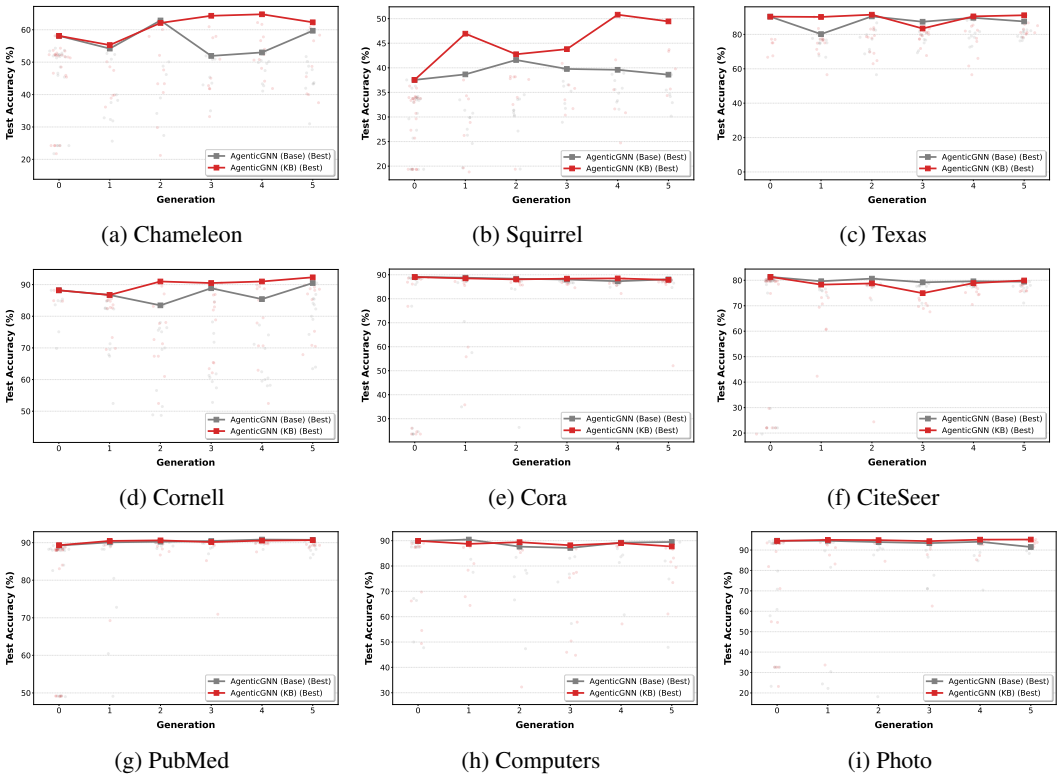


Figure 9: Heatmap of node-classification rank (2.5%:2.5%:95%).



(a) Chameleon (b) Squirrel (c) Texas

(d) Cornell (e) Cora (f) CiteSeer

(g) PubMed (h) Computers (i) Photo

Figure 10: Accuracy curves on nine benchmark datasets in knowledge-base ablation study.