A Appendices

A.1 Further details on spatial-based, spectral-based, and spectral-designed graph convolution

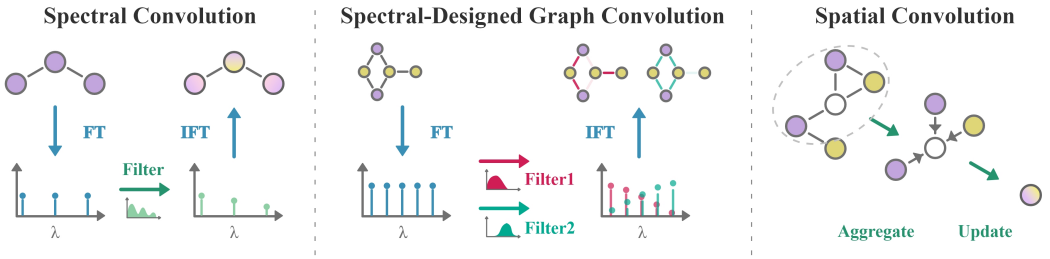

Figure 3: Left: Spectral Convolution transforms graph signals to the spectral domain via Graph Fourier Transform (Graph FT), applies filtering, and returns to the graph domain through Inverse Graph Fourier Transform (Graph IFT). Right: In Spatial Convolution, the target node (shown in white) aggregates information from its neighbors to update its representation. Center: Spectral-Designed Graph Convolution applies a set of filters in the spectral domain and then transforms back to the graph domain to obtain the kernels for spatial convolution.

Spatial-based graph convolutions are analogous to convolutions on images in the way that they both focus on local neighbors, as seen in Fig. 3 (right). In early research, Neural Network for Graphs (NN4G) [25] aggregates information from reachable neighbors by summing up their hidden states. GraphSage [26] then introduced random sampling of neighbors followed by an aggregation function.

Spectral-based graph convolutions are grounded in spectral graph theory. They operate in the Fourier domain by the eigendecomposition of the (normalized) Laplacian matrix $\mathbf{L}^{(\text{norm})}$. This method transforms signals on the graph into the spectral domain by applying the Graph Fourier Transform (Graph FT) defined as the product of the signal with the eigenvector matrix $\mathbf{U}^{\mathbf{T}}$ of $\mathbf{L}$. Also, various filters can be applied to the signals, these filtered signals can then be mapped back to the spatial domain through the Inverse Graph Fourier Transform (Graph IFT) with $\mathbf{U}$, as seen in Fig. 3 (left). Early methods like Spectral CNN [27] offer better global information capture but are computationally dense due to eigendecomposition. ChebNet [28] then reduces the complexity by approximating the eigendecomposition with a K-order Chebyshev polynomial, but also limits the receptive field to a K-order neighborhood. Inspired by this, GCN [29] limited the polynomial order to K=1, achieving an effective convolution layer with fewer parameters. Since GCN "works on local neighborhoods", it is also considered a spatial-based approach [30].

A.2 Spectral Responses

Taking advantage of the Spectral Parsing Module's ability to combine the outputs of multiple convolutional kernels, we employed all five filters designed in DSGCN, i.e., one low-pass, one high-pass, and three band-passes, whose frequency responses and expressions are shown in Fig. 4 and Fig. 5.

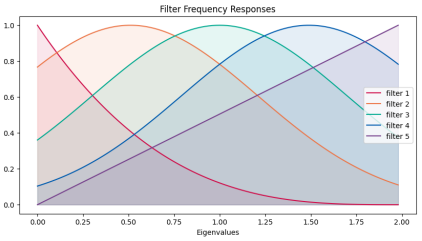

Figure 4: Frequency responses of each filter.

$$F_1(\lambda) = \left(1 - \frac{\lambda}{\lambda_{\max}}\right)^3, \quad \text{(Low-pass)}$$

$$F_2(\lambda) = e^{-((\lambda-\lambda_{c1})^2)\cdot\gamma}, \quad \text{(Band-pass 1)}$$

$$F_3(\lambda) = e^{-((\lambda-\lambda_{c2})^2)\cdot\gamma}, \quad \text{(Band-pass 2)}$$

$$F_4(\lambda) = e^{-((\lambda-\lambda_{c3})^2)\cdot\gamma}, \quad \text{(Band-pass 3)}$$

$$F_5(\lambda) = \frac{\lambda}{\lambda_{\max}}, \quad \text{(High-pass)}$$

Figure 5: Designed filters, where $\lambda_{c1}, \lambda_{c2}, \lambda_{c3}$ are the center frequencies and γ is the bandwidth parameter.

A.3 Further details on experimental settings

For regression tasks, the target values y_{true} are normalized to the range 0 to 1 using min-max normalization. Following the node regression protocol in CoEvoGNN [15], the evaluation metrics are RMSE (Root Mean Squared Error) and MAE (Mean Absolute Error). The final score is the average of the scores for each snapshot, with lower scores indicating better performance. To ensure reproducibility and assess variance, each experiment is run five times with different random seeds (0, 1, 2, 3, 4). These seeds are used to initialize the model parameters as well as the hidden states of the nodes for the first k time steps. For fairness in the experiments, the node embedding dimension d_h is uniformly set to 64 across all conducted experiments. We use the default hyperparameter of CoEvoGNN, marked as CoEvoGNN* in Table 2, and also apply the same hyperparameters for CoEvoGNN, EvolveGCN, and DspGNN as follows: learning rate = 0.003, weight decay = 1×10^{-6}, and $K = 6$, which means using the adjacency matrices and the encoded hidden states from the previous six snapshots to predict the hidden state of the next snapshot. The loss function is the same for all the models which inherited from CoEvoGNN. In this configuration, all the models converge well and have access to relatively rich information about the past, finally, they perform the regression task with a linear layer as in the protocol of CoEvoGNN.

A.4 Further details on Active Node Mapping Procedure

The concrete procedure of Active Node Mapping can be expressed as follows:

1. Map the indexes of the active nodes at time t from global node index 1 to N to a sequence of natural numbers from 1 to N_t. This mapping function is represented by a matrix $\mathbf{M}_t$ of dimensions $N \times N_t$ such that $\tilde{\mathbf{A}}_t = \mathbf{A}_t\mathbf{M}_t$. The post-mapping degree matrix and Laplacian matrix are also computed based on $\tilde{\mathbf{A}}_t$, denoted as $\tilde{\mathbf{D}}_t$ and $\tilde{\mathbf{L}}_t$, respectively.
2. Perform eigendecomposition on the reduced graph, denoted as $\tilde{\mathbf{L}}_t^{\text{norm}} = \tilde{\mathbf{U}}_t\tilde{\mathbf{\Lambda}}_t\tilde{\mathbf{U}}_t^{\mathbf{T}}$
3. Filter the eigenvalue matrix $\tilde{\mathbf{\Lambda}}$ through a function $g(\cdot)$.
4. Apply graph IFT to get the spectral-designed convolutional kernel(s) $\tilde{\mathbf{C}}_t = \tilde{\mathbf{U}}_t g(\tilde{\mathbf{\Lambda}}_t)\tilde{\mathbf{U}}_t^{\mathbf{T}}$
5. Apply the inverse mapping by multiplying the matrix $\mathbf{M^T}_t$ to the kernel $\tilde{\mathbf{C}}_t$ for mapping from the active node index back to the global node index for assigning the weights to corresponding edges.

A.5 Speedup of Eigendecomposition with Active Node Mapping

In this subsection, we provide a theoretical and empirical analysis of the computational cost for eigendecomposition with and without Active Node Mapping (ANM). The theoretical maximum space complexity without ANM is $O(N^2)$ and the time complexity is $O(N^3)$ (multiplied by T, the number of snapshots). In contrast, with ANM, the maximum space complexity is determined by the largest number of active nodes $O(max(N_t)^2)$, and the time complexity is reduced to the sum of the cubes of active nodes per snapshot, $\Sigma_t(N_t)^3$. Based on these theoretical formulas, we calculated the order-of-magnitude estimates for both space and time complexity as a theoretical reference (hence, no units are provided), listed in Table 3.

For the empirical evaluation, we performed five runs of eigendecomposition for all snapshots on Google Colab with consistent configurations, not including adjacency matrix generation and I/O time. The median time for each approach is reported in the table 3. W/O ANM implementations use either numpy.linalg.eigh [2] for the standard algorithm or scipy.sparse.linalg.eigs [3] optimized for sparse matrices. Both W/O ANM implementations have a runtime at least 100 times greater than our ANM approach across three datasets.

[2] https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig.html

[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.eigsh.html

Table 3: Computational cost comparison between without and with Active Node Mapping (ANM) techniques. 'Max space complexity' refers to the theoretical maximum space complexity required. 'Time complexity' refers to the computational operations based on theoretical time complexity. 'Real time consumed' shows the time taken to perform eigendecomposition on all snapshots. W/O ANM uses a standard numpy algorithm or scipy sparse-matrix optimized version, and ANM uses only the standard numpy algorithm. The value shown is the median value measured over five runs.

		W/O Active Node Mapping		With Active Node Mapping
Max space complexity (Theo.)	Bitcoin-Alpha		1×10^7	3×10^5
	Bitcoin-OTC		3×10^7	3×10^5
	MovieLens-100K		1×10^8	6×10^6
Time complexity (Theo.)	Bitcoin-Alpha		7×10^{12}	6×10^8
	Bitcoin-OTC		3×10^{13}	1×10^9
	MovieLens-100K		9×10^{13}	1×10^{11}
Real time consumed (seconds)	Library	Numpy	Scipy	Numpy
	Bitcoin-Alpha	1826.31	225.69	0.31
	Bitcoin-OTC	7203.62	592.23	0.55
	MovieLens-100K	11150.19	1159.91	12.83