A. Extra Details on Model Architectures

In the propagation layers of the graph embedding and matching models, we used an MLP with one hidden layer as the f_{message} module, with a ReLU nonlinearity on the hidden layer. For node state vectors (the $\mathbf{h}_i^{(t)}$ vectors) of dimension D, the size of the hidden layer and the output is set to $2D$. We found it to be beneficial to initialize the weights of this f_{message} module to be small, which helps stablizing training. We used the standard Glorot initialization with an extra scaling factor of 0.1. When not using this small scaling factor, at the begining of training the message vectors when summed up can have huge scales, which is bad for learning.

One extra thing to note about the propagation layers is that we can make all the propagation layers share the same set of parameters, which can be useful if this is a suitable inductive bias to have.

We tried different f_{node} modules in both experiments, and found GRUs to generally work better than one-hidden layer MLPs, and all the results reported uses GRUs as f_{node}, with the sum over edge messages $\sum_j \mathbf{m}_{j\to i}$ treated as the input to the GRU for the embedding model, and the concatenation of $\sum_j \mathbf{m}_{j\to i}$ and $\sum_{j'} \boldsymbol{\mu}_{j'\to i}$ as the input to the GRU for the matching model.

In the aggregator module, we used a single linear layer for the node transformation MLP and the gating MLP_{gate} in Eq.3. The output of this linear layer has a dimensionality the same as the required dimensionality for the graph vectors. $\sigma(x) = \frac{1}{1+e^{-x}}$ is the logistic sigmoid function, and $\odot$ is the element-wise product. After the weighted sum, another MLP with one hidden layers is used to further transform the graph vector. The hidden layer has the same size as the output, with a ReLU nonlinearity.

For the matching model, the attention weights are computed as

$$a_{j\to i} = \frac{\exp(s_h(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}))}{\sum_{j'} \exp(s_h(\mathbf{h}_i^{(t)}, \mathbf{h}_{j'}^{(t)}))}. \tag{16}$$

We have tried the Euclidean similarity $s_h(\mathbf{h}_i, \mathbf{h}_j) = -\|\mathbf{h}_i - \mathbf{h}_j\|^2$ for s_h, as well as the dot-product similarity $s_h(\mathbf{h}_i, \mathbf{h}_j) = \mathbf{h}_i^\top \mathbf{h}_j$, and they perform similarly without significant difference.

B. Extra Experiment Details

We fixed the node state vector dimensionality to 32, and graph vector dimensionality to 128 throughout both the graph edit distance learning and binary function similarity search tasks. We tuned this initially on the function similarity search task, which clearly performs better than smaller models. Increasing the model size however leads to overfitting for that task. We directly used the same setting for the edit distance learning task without further tuning. Using larger models there should further improve model performance.

B.1. Learning Graph Edit Distances

In this task the nodes and edges have no extra features associated with them, we therefore initialized the $\mathbf{x}_i$ and $\mathbf{x}_{ij}$ vectors as vectors of 1s, and the encoder MLP in Eq.1 is simply a linear layer for the nodes and an identity mapping for the edges.

We searched through the following hyperparameters: (1) triplet vs pair training; (2) number of propagation layers; (3) share parameters on different propagation layers or not. Learning rate is fixed at 0.001 for all runs and we used the Adam optimizer (Kingma & Ba, 2014). Overall we found: (1) triplet and pair training performs similarly, with pair training slightly better, (2) using more propagation layers consistently helps, and increasing the number of propagation layers T beyond 5 may help even more, (3) sharing parameters is useful for performance more often than not.

Intuitively, the baseline WL kernel starts by labeling each node by its degree, and then iteratively updates a node's representation as the histogram of neighbor node patterns, which is effectively also a graph propagation process. The kernel value is then computed as a dot product of graph representation vectors, which is the histogram of different node representations. When using the kernel with T iterations of computation, a pair of graphs of size $|V|$ can have as large as a $2|V|T$ dimensional representation vector for each graph, and these sets of effective 'feature' types are different for different pairs of graphs as the node patterns can be very different. This is an advantage for WL kernel over our models as we used a fixed sized graph vector regardless of the graph size. We evaluate WL kernel for T up to 5 and report results for the best T on the evaluation set.

In addition to the experiments presented in the main paper, we have also tested the generalization capabilities of the proposed models, and we present the extra results in the following.

Train on small graphs, generalize to large graphs. In this experiment, we trained the GSL models on graphs with n sampled uniformly from 20 to 50, and p sampled from range $[0.2, 0.5]$ to cover more variability in graph sizes and edge density for better generalization, and we again fix $k_p = 1, k_n = 2$. For evaluation, we tested the best embedding models and matching models on graphs with $n = 100, 200$ and $p = 0.2, 0.5$, with results shown in Table 3. We can see that for this task the GSL models trained on small graphs can generalize to larger graphs than they are trained on. The performance falls off a bit on much larger graphs with much more nodes and edges. This is also partially caused

Eval Graphs	WL kernel	GNN	GMN
$n = 100, p = 0.2$	98.5 / 99.4	96.6 / 96.8	96.8 / 97.7
$n = 100, p = 0.5$	86.7 / 97.0	79.8 / 81.4	83.1 / 83.6
$n = 200, p = 0.2$	99.9 / 100.0	88.7 / 88.5	89.4 / 90.0
$n = 200, p = 0.5$	93.5 / 99.2	72.0 / 72.3	68.3 / 70.1

Table 3. Generalization performance on large graphs for the GSL models trained on small graphs with $20 \leq n \leq 50$ and $0.2 \leq p \leq 0.5$.

by the fact that we are using a fixed sized graph vector throughout the experiments , but the WL kernel on the other hand has much more effective 'features' to use for computing similarity. On the other hand, as shown before, when trained on graphs from distributions we care about, the GSL models can adapt and perform much better.

Train on some k_p, k_n combinations, test on other combinations. We have also tested the model trained on graphs with $n \in [20, 50]$, $p \in [0.2, 0.5]$, $k_p = 1, k_n = 2$, on graphs with different k_p and k_n combinations. In particular, when evaluated on $k_p = 1, k_n = 4$, the models perform much better than on $k_p = 1, k_n = 2$, reaching 1.0 AUC and 100% triplet accuracy easily, as this is considerably simpler than the $k_p = 1, k_n = 2$ setting. When evaluated on graphs with $k_p = 2, k_n = 3$, the performance is workse than $k_p = 1, k_n = 2$ as this is a harder setting.

In addition, we have also tried training on the more difficult setting $k_p = 2, k_n = 3$, and evaluate the models on graphs with $k_p = 1, k_n = 2$ and $n \in [20, 50], p \in [0.2, 0.5]$. The performance of the models on these graphs are actually be better than the models trained on this setting of $k_p = 1, k_n = 2$, which is surprising and clearly demonstrates the value of good training data. However, in terms of generalizing to larger graphs models trained on $k_p = 2, k_n = 3$ does not have any significant advantages.

B.2. Binary Function Similarity Search

In this task the edges have no extra features so we initialize them to constant vectors of 1s, and the encoder MLP for the edges is again just an identity mapping. When using the CFG graph structure only, the nodes are also initialized to constant vectors of 1s, and the encoder MLP is a linear layer. In the case when using assembly instructions, we have a list of assembly code associated with each node. We extracted the operator type (e.g. `add`, `mov`, etc.) from each instruction, and then embeds each operator into a vector, the initial node representation is a sum of all operator embeddings.

We searched through the following hyperparameters: (1) triplet or pair training, (2) learning rate in $\{10^{-3}, 10^{-4}\}$, (3) number of propagation layers; (4) share propagation layer parameters or not; (5) GRU vs one-layer MLP for the f_{node} module.

Overall we found that (1) triplet training performs slightly better than pair training in this case; (2) both learning rates can work but the smaller learning rate is more stable; (3) increasing number of propagation layers generally helps; (4) using different propagation layer parameters perform better than using shared parameters; (5) GRUs are more stable than MLPs and performs overall better.

In addition to the results reported in the main paper, we have also tried the same models on another dataset obtained by compiling the compression software `unrar` with different compilers and optimization levels. Our graph similarity learning methods also perform very well on the unrar data, but this dataset is a lot smaller, with around 400 functions only, and overfitting is therefore a big problem for any learning based model, so the results on this dataset are not very reliable to draw any conclusions.

A few more control-flow graph examples are shown in Figure 5. The distribution of graph sizes in the training set is shown in Figure 6.

C. Extra Attention Visualizations

A few more attention visualizations are included in Figure 7, Figure 8 and Figure 9. Here the graph matching model we used has shared parameters for all the propagation and matching layers and was trained with 5 propagation layers. Therefore we can use a number T different from the number of propagation layers the model is being trained on to test the model's performance. In both visualizations, we unrolled the propagation for up to 9 steps and the model still computes sensible attention maps even with $T > 5$.

Note that the attention maps do not converge to very peaked distributions. This is partially due to the fact that we used the node state vectors both to carry information through the propagation process, as well as in the attention mechanism as is. This makes it hard for the model to have very peaked attention as the scale of these node state vectors won't be very big. A better solution is to compute separate key, query and value vectors for each node as done in the tensor2tensor self-attention formulation (Vaswani et al., 2017), which may further improve the performance of the matching model.

Figure 7 shows another possibility where the attention maps do not converge to very peaked distributions because of in-graph symmetries. Such symmetries are very typical in graphs. In this case even though the attention maps are not peaked, the cross graph communication vectors μ are still zero, and the two graphs will still have identical representation vectors.

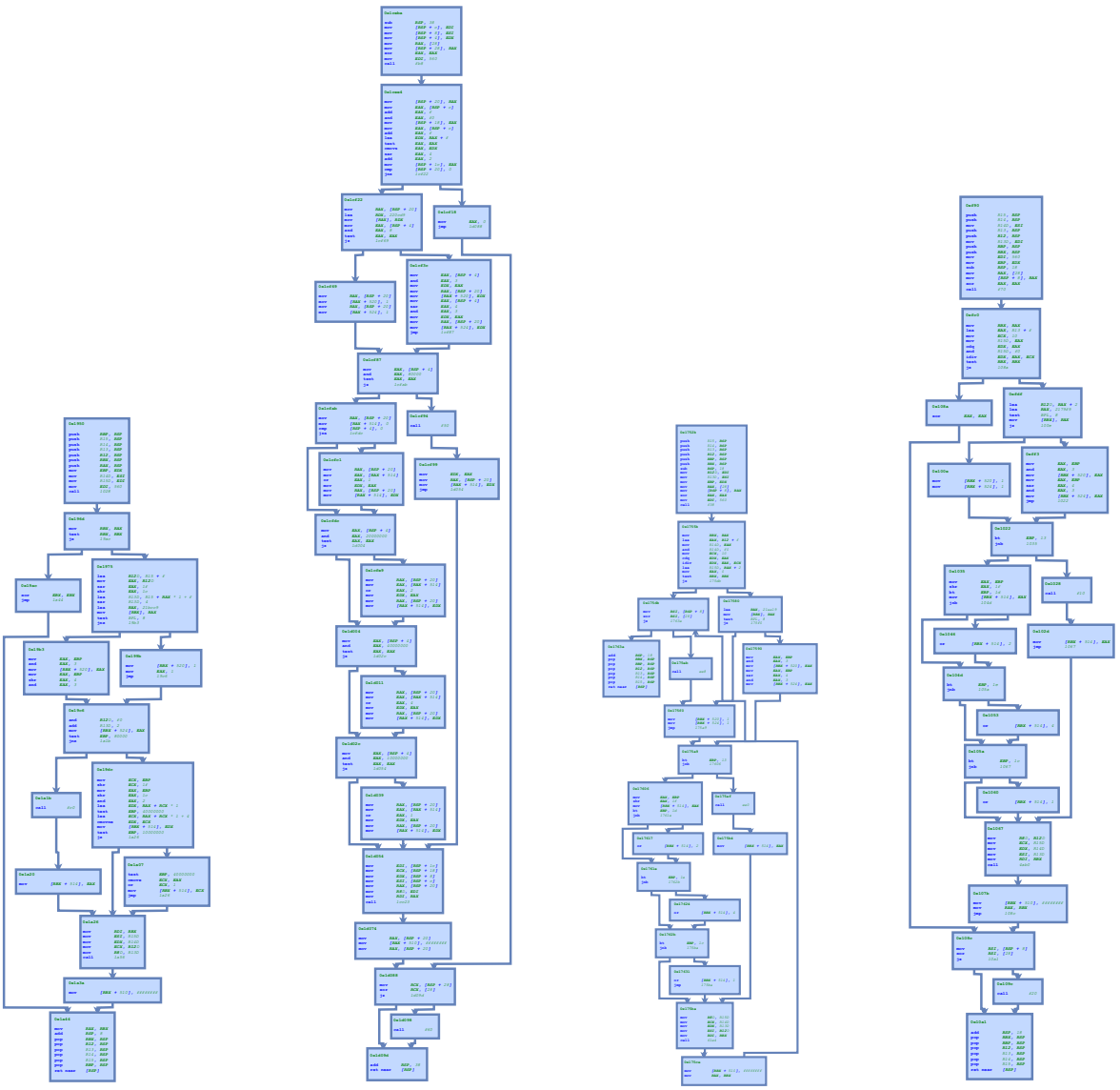

Figure 5. Example control flow graphs for the same binary function, compiled with different compilers (`clang` for the leftmost one, `gcc` for the others) and optimization levels. Note that each node in the graphs also contains a set of assembly instructions which we also take into account when computing similarity using learned features.

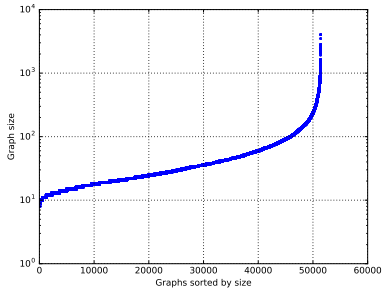

Figure 6. Control flow graph size distribution in the training set. In this plot the graphs are sorted by size on the x axis, each point in the figure corresponds to the size of one graph.

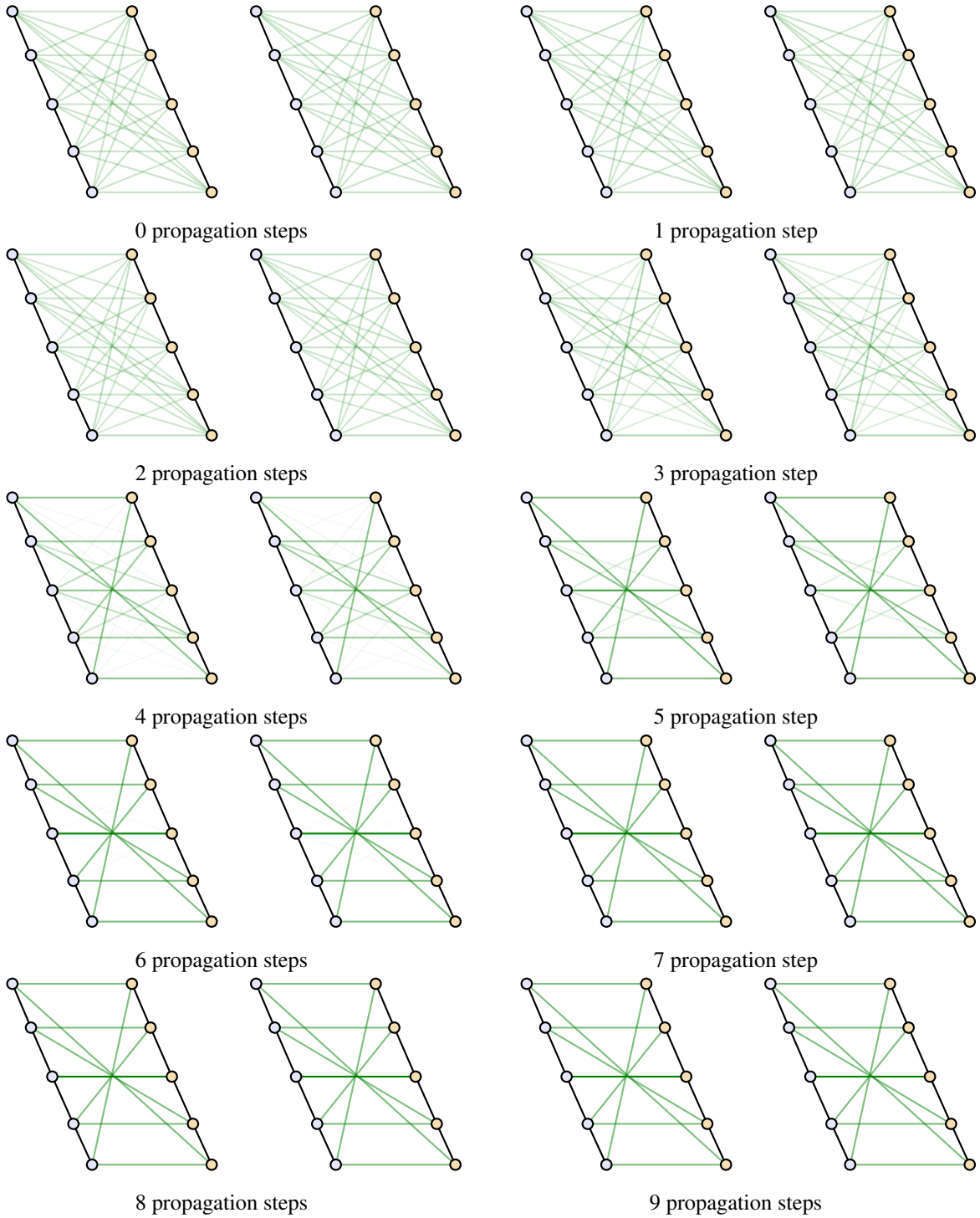

Figure 7. The change of cross-graph attention over propagation layers. Here the two graphs are two isomorphic chains and there are some in-graph symmetries. Note that in the end the nodes are matched to two corresponding nodes with equal weight, except the one at the center of the chain which can only match to a single other node.

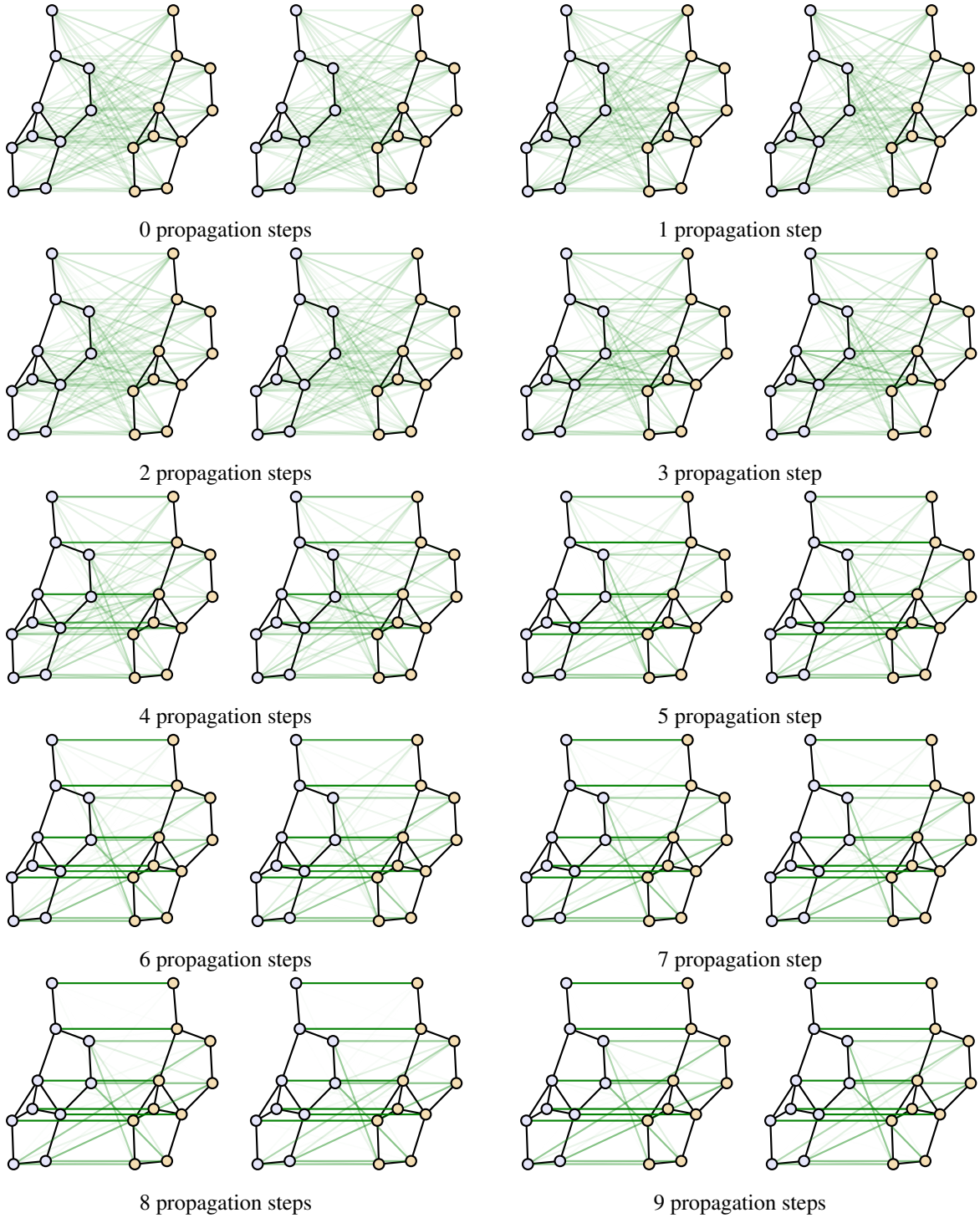

Figure 8. The change of cross-graph attention over propagation layers. Here the two graphs are isomorphic, with graph edit distance 0. Note that in the end a lot of the matchings concentrated on the correct match.

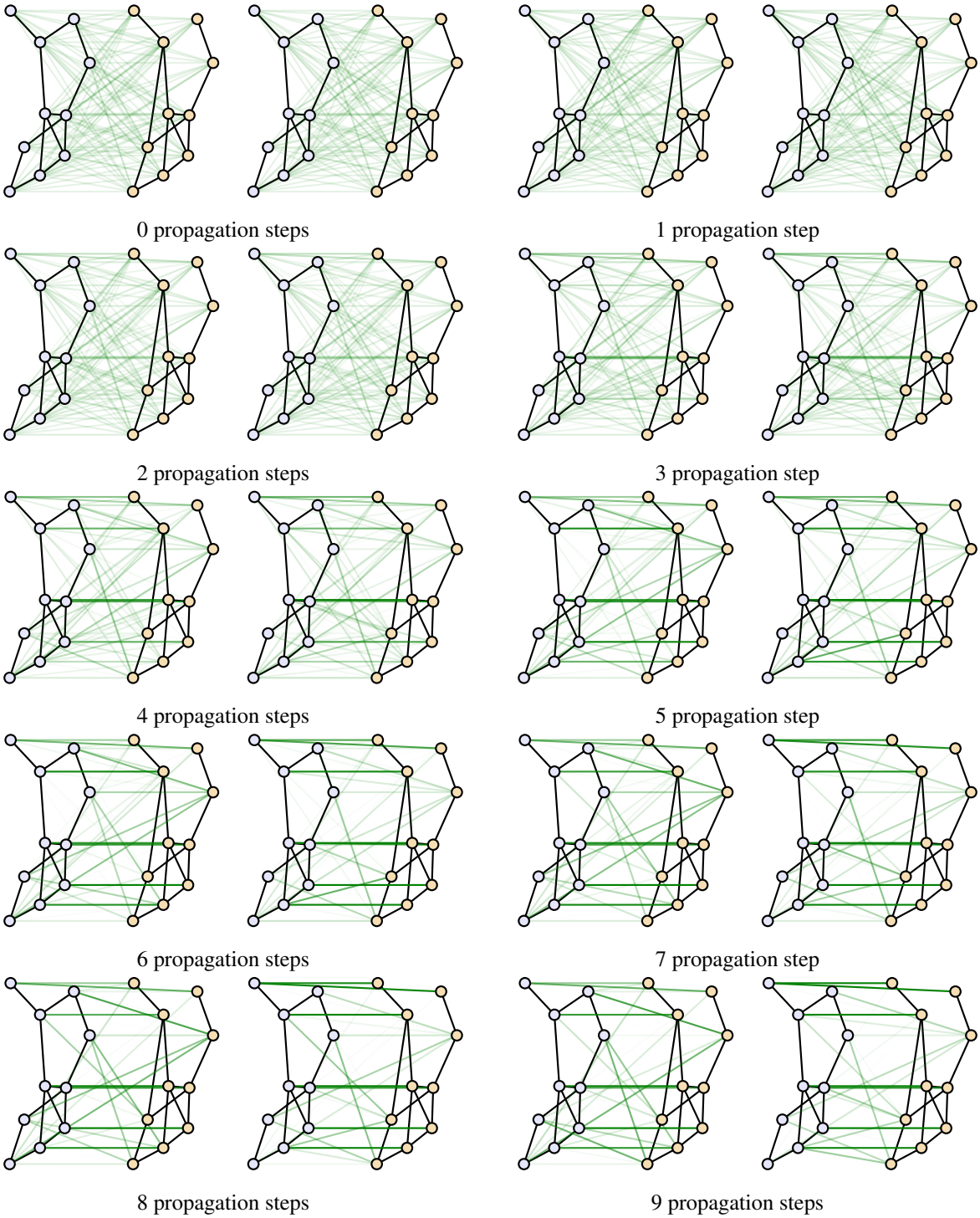

Figure 9. The change of cross-graph attention over propagation layers. The edit distance between these two graphs is 1.