GLSearch: Maximum Common Subgraph Detection via Learning to Search

Yunsheng Bai [*1] **Derek Xu** [*1] **Yizhou Sun** [1] **Wei Wang** [1]

A. Dataset Description

This section describes the datasets used for evaluating our model and baselines. Section B.1 describes the dataset we use for training GLSEARCH as well as baseline learning based graph matching models.

We use the following real-world datasets for evaluation:

- NCI109: It is a collection of small-sized chemical compounds (Wale et al., 2008) whose nodes are labeled indicating atom type. We form 100 graph pairs from the dataset whose average graph size (number of nodes) is 28.73.

- ROAD: The graph is a road network of California whose nodes indicate intersections and endpoints and edges represent the roads connecting the intersections and endpoints (Leskovec et al., 2009). The graph contains 1965206 nodes, from which we randomly sample a connected subgraph of around 0.05% nodes twice to generate two subgraphs for the graph pair $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$.

- DBEN, DBZH, and DBPD: It is a dataset originally used in a work on cross-lingual entity alignment (Sun et al., 2017). The dataset contains pairs of DBpedia knowledge graphs in different languages. For DBEN, we use the English knowledge graph and sample 10% nodes twice to generate two graphs for our task. For DBZH, we sample around 10% nodes from the knowledge graph in Chinese. For DBPD, we sample once from the English graph to get $\mathcal{G}_1$ and sample once from the Chinese graph to get $\mathcal{G}_2$. Note that although the nodes have features, we do not use them because our task is more about graph structural matching rather than node semantic meanings, and leave the incorporation of continuous node initial representations as future work.

- ENRO: The graph is an email communication network whose nodes represent email addresses and (undirected) edges represent at least one email sent between the addresses (Klimt & Yang, 2004). From the total 36692 nodes, we sample around 10% nodes to generate the graph pair.

- COPR: An Amazon computer product network whose nodes represent goods and edges represent two goods frequently purchased together (Shchur et al., 2018). The graph contains 703655 nodes from which we sample around 0.5% to get the pair we use.

- CIRC: This is a graph pair where each graph is a circuit diagram whose nodes represent devices/wires and edges represent the connecting relations between devices and wires. In other words, each node is either a device or a wire, and the entire graph is bipartite. The two graphs given are known to be **isomorphic**[1] and we do not perform any sampling. Nodes have labels about the type of the device/wire. In real world, the successful matching of circuit layout diagrams is an essential process in circuit design verification.

- HPPI: It is a human protein-protein interaction network whose nodes represent proteins and edges represent physical interaction between proteins in a human cell (Agrawal et al., 2018). From the 21557 nodes, we sample around 10% nodes to generate the pair used in experiments.

- ROAD-CA: We use the same road network of California as the ROAD dataset, but this time, from the 1965206 nodes, we randomly sample a connected subgraph of around 50.0% nodes twice to generate two subgraphs for the graph pair.

- ROAD-TX: Similar to ROAD-CA, but the road network is in Texas (Leskovec et al., 2009). The graph contains 1379917 nodes, from which we randomly sample a connected subgraph of around 80% nodes twice to generate two subgraphs for the graph pair.

The details of all the graph pairs can be found in Table 1.

[*]Equal contribution [1]Department of Computer Science, University of California, Los Angeles, California, USA. Correspondence to: Yunsheng Bai <yba@ucla.edu>, Derek Xu <derekqxu@ucla.edu>.

Proceedings of the 38th International Conference on Machine Learning, PMLR 139, 2021. Copyright 2021 by the author(s).

[1]Section E.3 shows results on synthetic datasets where the MCS size lower bound known.

Table 1: Details of real-world graph pairs used in evaluating the performance of baseline methods and GLSEARCH.

Name	Description	$\lvert\mathcal{V}_1\rvert$	$\lvert\mathcal{V}_2\rvert$	$\lvert\mathcal{E}_1\rvert$	$\lvert\mathcal{E}_2\rvert$	$\frac{\lvert\mathcal{E}_1\rvert}{\lvert\mathcal{V}_1\rvert}$	$\frac{\lvert\mathcal{E}_2\rvert}{\lvert\mathcal{V}_2\rvert}$
ROAD	Road Network	1114	652	1454	822	1.305	1.261
DBEN	Knowledge Graph	1945	1945	6242	5851	3.209	3.008
DBZH	Knowledge Graph	1907	1907	4856	4948	2.546	2.595
DBPD	Knowledge Graph	1945	1907	6242	4856	3.209	2.546
ENRO	Email Communication Network	3369	3369	46399	50637	13.772	15.030
COPR	Product Co-purchasing Network	3518	3518	56028	40633	15.926	11.550
CIRC	Circuit Layout Diagram	4275	4275	6128	6128	1.433	1.433
HPPI	Protein-Protein Interaction Network	2152	2152	54910	54132	25.516	25.154
ROAD-CA	Road Network	978513	978513	1404115	1366917	1.435	1.397
ROAD-TX	Road Network	1080909	1080909	1503531	1507440	1.391	1.395

For synthetic datasets, we generate graph pairs using the Barabási-Albert (BA) (Barabási & Albert, 1999) algorithm (edge density set to 5), the Erdős-Rényi (ER) (Gilbert, 1959) algorithm (edge density set to 0.08), and the Watts–Strogatz (WS) (Watts & Strogatz, 1998) algorithm (rewiring probability set to 0.2 and ring density set to 4), respectively.

B. Details on DQN and Training GLSEARCH

B.1. Training Data Preparation: Curriculum Learning

Curriculum learning (Bengio et al., 2009) is a strategy for training machine learning models whose core idea is to train a model first using "easy" examples before moving on to using "hard" ones. Our goal is to train a general model for MCS detection task which works well on general testing graph pairs from different domains. Therefore, we employ the idea of curriculum learning in training our GLSEARCH. More specifically, we prepare the training graph pairs in the following way:

- Curriculum 1: The first curriculum consists of the easiest graph pairs that are small: (1) We sample 30 graph pairs from AIDS (Zeng et al., 2009), a chemical compound dataset usually for graph similarity computation (Bai et al., 2019) where each graph has less than or equal to 10 nodes; (2) We sample 30 graph pairs from LINUX (Wang et al., 2012), another dataset commonly used for graph matching consisting of small program dependency graphs generated from Linux kernel; (3) So far we have 60 real-world graph pairs. We then generate 60 graph pairs using popular graph generation algorithms. Specifcally, we generate 20 graph pairs using the BA algorithm, 20 graph pairs using the ER algorithm, and 20 graph pairs using the Watts–Strogatz WS algorithm, respectively. Details of the graphs can be found in Table 2. In summary, the first curriculum contains 120 graph pairs in total.
- Curriculum 2: After the first curriculum, each next curriculum contains graphs that are larger and harder to match than the previous curriculum. For the second curriculum, we sample 30 graph pairs from PTC (Shrivastava & Li, 2014), a collection of chemical compounds, 30 graph paris from IMDB (Yanardag & Vishwanathan, 2015), a collection of ego-networks of movie actors/actresses, and generate 20 graph pairs again using the BA, ER, and WS algorithms but with larger graph sizes.
- Curriculum 3: For the third curriculum, we sample 30 graph pairs from MUTAG (Debnath et al., 1991), a collection of chemical compounds, 30 graph paris from REDDIT (Yanardag & Vishwanathan, 2015), a collection of ego-networks corresponding to online discussion threads, and generate 20 even larger graph pairs using the BA, ER, and WS algorithms.
- Curriculum 4: For the last curriculum, we sample 30 graph pairs from WEB (Riesen & Bunke, 2008), a collection of text document graphs, 30 graph paris from MCSPLAIN-CONNECTED (McCreesh et al., 2017), a collection of synthetic graph pairs adopted by MCSP, and generate 20 graph pairs again using BA, ER, and WS algorithms but with larger graph sizes.

For each curriculum, we train the model for 2500 iterations before moving on to the next, resulting in 10000 training iterations in total.

B.2. Training Techniques and Details

B.2.1. STAGE 1: PRE-TRAINING

For the first 1250 iterations, we pre-train our DQN with the supervised true target y_t obtained as follows:

- For each graph pair, we run the complete search, i.e. we do not perform any pruning for unpromising states. The entire search space is explored, and the future

Table 2: Training graph details. For synthetic graphs, "ed", "p", and "rd" represent edge density, rewiring probability, and ring density, respectively.

Curriculum	Data Source	# Pairs
Curriculum 1	AIDS	30
	LINUX	30
	BA:n=16,ed=5	20
	ER:n=14,ed=0.14	20
	WS:n=18,p=0.2,rd=2	20
Curriculum 2	PTC	30
	IMDB	30
	BA:n=32,ed=4	20
	ER:n=30,ed=0.12	20
	WS:n=34,p=0.2,rd=2	20
Curriculum 3	MUTAG	30
	REDDIT	30
	BA:n=48,ed=4	20
	ER:n=46,ed=0.1	20
	WS:n=50,p=0.2,rd=4	20
Curriculum 4	WEB	30
	MCSPLAIN-CONNECTED	30
	BA:n=62,ed=3	20
	ER:n=64,ed=0.08	20
	WS:n=66,p=0.2,rd=4	20

reward for every action can be found by finding the longest path starting from the action to a terminal state. Since graphs are small in the initial stage, such complete search can be affordable. Using Figure 2 in the main text as an example, for the action that causes state 0 to transition to state 6, the longest path is 0, 6, 7, 11, 12, 13 (or 0, 6, 7, 11, 12, 14).

- Given the longest path found for each action, we then compute $y_t = 1 + \gamma + \gamma^2 + ... + \gamma^{(L-1)}$, where γ is the discount factor set to 1.0, L is the length of the longest path. In the example above, $y_t = 5$, intuitively meaning that at state 0, for the action that leads to state 6, in future the best solution will have 5 more nodes. In contrast, the action 0 to 1 has $y_t = 4$, meaning the action 0 to 6 is more preferred.
- Given the true target computed for each action, we run the mini-batch gradient descents over the mse loss $(y_t - Q(s_t, a_t))^2$, where the batch size (number of sampled actions) is set to 32.

B.2.2. STAGE 2: IMITATION LEARNING AND STAGE 3

For stage 2 (2500 iterations) and stage 3 (6250 iterations), we train the DQN using the framework proposed in Mnih et al. (2013). The difference is that in stage 2, instead of allowing the model to use its own predicted $Q(s_t, a_t)$ at each state, we let the model make a decision using the heuristics by the MCSP algorithm, which serves as an expert providing trajectories in stage 2. We aim to outperform MCSP eventually after training using our own predicted $Q(s_t, a_t)$ in stage 3.

Here we describe the procedure of the training process. In each training iteration, we sample a graph pair from the current curriculum for which we run the DQN multiple times until a terminal state is reached to collect all the transitions, i.e. 4-tuples in the form of (s_t, a_t, r_t, s_{t+1}) where r_t is 1 and $y_t = 1 + \gamma \max_{a'} Q(s_{t+1}, a')$, and store them into a global experience replay buffer, a queue that maintains the most recent L 4-tuples. In our calculations, $L = 1024$. Afterwards, at the end of the iteration, the agent gets updated by performing the mini-batch gradient descents over the mse loss $(y_t - Q(s_t, a_t))^2$, where the batch size (number of sampled transitions from the replay buffer) is set to 32.

To stabilize our training, we adopt a target network which is a copy of the DQN network and use it for computing $\max_{a'} \gamma Q(s_{t+1}, a')$. This target network is synchronized with the DQN periodically, in every 100 iterations.

Since at the beginning of stage 3, the Q approximation may still be unsatisfactory, and random behavior may be better, we adopt the epsilon-greedy method by switching between random policy and Q policy using a probability hyperparameter ϵ. Thus, the decision is made as $\arg\max \boldsymbol{q}$ where $\boldsymbol{q}$ is the our predicted $Q(s_t, a_t)$ for all possible actions (i, j) with $1 - \epsilon$ probability; With ϵ probability, the decision is random. This probability is tuned to decay slowly as the agent learns to play the game, eventually stabilizing at a fixed probability. We set the starting epsilon to 0.1 decaying to 0.01.

B.3. DQN Parameter Details

In experiments, we use SUM followed by an MLP for READOUT and 1DCONV+MAXPOOL followed by an MLP for INTERACT. Specifically, the MLP has 2 layers down-projecting the node embeddings from 64 to 32 dimensions. Notice that different types of embeddings require different MLPs, e.g. the MLP used for aggregating and generating graph-level embeddings is different from the MLP used for aggregating and generating subgraph-level embeddings.

For 1DCONV+MAXPOOL, we apply a 1-dimensional convolutional neural network to each one of two embeddings being interacted, followed by performing max pooling across each dimension in the two embeddings before feeding into an MLP to generate the final interacted embedding. Specifically, the 1DCONV contains a filter of size 3 and stride being 1. The MLP afterwards is again a 2-layer MLP projecting the dimension to 32. As shown in the main text, such learnable interaction operator brings performance gain compared to simple summation based interaction.

The final MLP takes four components, $\boldsymbol{h}_{\mathcal{G}} = \text{INTERACT}(\boldsymbol{h}_{\mathcal{G}_1}, \boldsymbol{h}_{\mathcal{G}_2})$, $\boldsymbol{h}_s = \text{INTERACT}(\boldsymbol{h}_{s1}, \boldsymbol{h}_{s2})$, $\boldsymbol{h}_{\mathcal{D}c}$, and $\boldsymbol{h}_{\mathcal{D}0}$, each with dimension 32. It consists of 7 layers down-projecting the 128-dimensional input (32×4) to a scalar as the predicted q score. For every MLP used in experiments, all the layers except the last use the $\mathrm{ELU}(x)$ activation function. An exception is the final MLP, whose last layer uses the $\mathrm{ELU}(x) + 1$ as the activation function to ensure positive q output.

A subtle point to notice is the necessity of using either nonlinear readout or nonlinear interaction for generating the bidomain representation. Otherwise, if both operators are a simple summation, the representation for all the connected bidomains ($\mathbf{h}_{\mathcal{D}c}$) is essentially the global summation of all nodes in all the connected bidomains. In other words, the nonlinearity of MLP in the readout operation or the interaction operator allows our model to capture the bidomain partitioning information in $\mathbf{h}_{\mathcal{D}c}$.

C. Notes on Search

C.1. Comparison with McSp and McSp+RL

The key idea of our model is that under a limited search budget, by exploring the most promising node pairs first, search can reach a larger common subgraph solution faster. In other words, for small graph pairs, all baseline models would obtain the exact MCS result as long as the search algorithm runs to complete, i.e. the stack is eventually empty, meaning no more actions to select and no more states to backtrack to (all states have been visited and fully expanded to all possible next states).

However, for large graph pairs, the task is NP-hard, and the complete search becomes nearly impossible. Exceptions exist though: For example, if the pruning condition based on the upper bound estimation is powerful enough to prune many states, the search may finish in relatively few iterations. However, we observe that the state-of-the-art solvers, McSp and McSp+RL, cannot finish completely for all the graph pairs used in testing. Instead of trying to improve the upper bound estimation to be more exact, in this paper, our goal is to learn a better node pair selection policy, to replace the heuristics used by baseline solvers.

Notice that our focus on node pair selection policy instead of upper bound estimation implies that a better selection policy would mean the search can quickly find a larger solution and update its best solution found so far $maxSol$. This not only mean when the search budget is used up, the result returned is larger, but also mean that for subsequent iterations (before the iteration limit is reached), more states would be pruned by checking $UB_t \leq |maxSol|$, thus further helping the search. In summary, in our framework, the upper bound computation strategy remains unchanged, yet the successful node pair selection policy benefits the search in two major ways.

Since we use McSp in the imitation learning stage of training our DQN, and compare with McSp and McSp+RL in the main text, we describe their node pair selection heuristics. For McSp, when entering a new state, it first selects the node with the largest node degree in $\mathcal{G}_1$, and then enumerates through all the nodes in $\mathcal{G}_2$ in descending order of node degrees. In the original implementation provided by McSp, this is achieved by recursive function calls. After all the nodes in $\mathcal{G}_2$ are visited, i.e. the depth-first search of all the node pairs (i, j) finishes where i is the largest-degree node in $\mathcal{G}_1$ and j is every node in $\mathcal{G}_2$, the algorithm selects the second largest-degree node in $\mathcal{G}_1$, and repeats the enumeration of nodes in $\mathcal{G}_2$. After all node pairs are exhausted, the function returns and the algorithm essentially backtracks to the parent state. If the current state is the root node in the search tree, the search is complete and the exact MCS is returned. However, as noted earlier, for large graph pairs it is almost impractical to search exhaustively and a budget on the amount of search conducted has to be applied. Thus, which node pairs to visit first matters a lot for successfully extracting a large solution for large input graphs. However, as seen in Figures 7 and 8, in many cases the true MCS does not contain large-degree nodes, since large-degree nodes tend to form more complicated subgraphs which are harder to match in the other input graph compared to simpler subgraphs like a chain. Thus, by visiting large-degree nodes first, McSp may not always yield a large solution fast.

In contrast, McSp+RL maintains a promising score for each node and iteratively updates the scores as search visits more states. The update formula is based on the reduction of upper bound for search, where upper bound in an overestimation of future subgraph size. As search makes progress, the scores are updated in each iteration, and nodes which cause large reduction in upper bound computation get large reward. This has the limitation that for each new graph pair, the scores associated with each node must be re-initialized to 0 and re-learned, since there is no neural network and the only learnable parameters are the scores for each node. At the beginning of search, all scores are initialized to 0, and the search has to break the tie using another heuristic, while once trained, our GLSearch can be applied to any new testing pair, and at the beginning of search, the learned parameters in GLSearch starts to benefit the search. In other words, the whole design of McSp+RL can be regarded as a search framework with shallow learning (without neural networks or training via back-propagation). GLSearch is the first model to use deep learning for node pair selection.

Another limitation of McSp+RL is that the scores maintained for nodes reflect the potential ability of a node to reduce the upper bound for future iterations in the current search, which is indirect as the MCS aims to find the largest common subgraph, not the reduction of upper bound. Moreover, the upper bound itself is an overestimation of future subgraph size, which may or may not be close enough to the actual best future subgraph size. In contrast, we aim to predict the q score for actions which directly reflect the best future subgraph size. Overall, the lack of deep learning ability causes McSp+RL not only to re-estimate the scores for the nodes for each new graph pair, but also to resort to the upper bound heuristic for updating the scores.

To ensure the budget on search iterations is applied consistently for all the models evaluated in the main text, we adapt the original recursive implementation of McSp in C++ to an iterative implementation in Python so that all the models compare with each other in the same programming language and the same search backbone algorithm. To be specific, we check the iteration count at the beginning of every search iteration and early stop the search if the pre-defined budget is reached.

C.2. Tree vs Sequence

At this point, having illustrated the differences between GLSearch and McSp and McSp+RL, it is worth clarifying whether GLSearch search yields a tree or sequence in different stages. For training stage 1 and 2, as described in Section B.2, our model is just randomly initialized and not well trained, so the pre-training and imitation learning stages use the policy of McSp instead of using its own predicted q scores. In stage 1, the complete search is performed to provide maximum amount of supervised signals, i.e. y_t, but in stage 2, we start using the RL training framework, i.e. experience replay buffer, target network, etc, so we run the agent multiple times until a terminal state is reached, corresponding to a sequence in a tree, which starts from the root node and ends at a leaf node. For stages 2 and 3, since sequences are generated instead of trees, for each sequence, the upper bound check always passes, because the pruning only happens when backtracking is allowed, i.e. a tree is formed. To see this more clearly, recall the pruning only happens if $UB_t \leq |maxSol|$ in Algorithm 1 in the main text. However, during the sequence generation process, $maxSol$ keeps increasing by one each time a new state is reached. Since $UB_t \leftarrow |curSol|$ + overestimate(s_t), $UB_t > |curSol|$ for non-terminal s_t, and $|curSol| = |maxSol|$, and thus $UB_t > |maxSol|$, and thus the pruning never happens.

At the beginning of stage 2, the sequences we collect are usually not long, since the policy is the same as McSp in stage 2. However, in stage 3, we start using our predicted $Q(s_t, a_t)$ to get such sequences, and at the end of training, when we apply GLSearch to testing pairs during inference, as shown in the main text, we perform better than McSp and all the other baselines. In inference, we completely rely on our predicted q scores as the policy, and a search tree is yielded, although the tree is not complete since the graph pairs are large and a search budget is reached.

C.3. Terminal Conditions

This section discusses on how a terminal state, i.e. a leaf node in the search tree, is determined. Notice our definition of bidomain (equivalence class) does not include node labels, and in each iteration, we allow the matching between nodes in the same bidomain with the same node label. We consider the node labels as additional pruning on each bidomain, i.e. we further only allow nodes with the same label to match within bidomain when considering actions to be fed into DQN. Suppose $\mathcal{G}_1$ and $\mathcal{G}_2$ are connected graphs. There are two cases:

- Case 1: Nodes are unlabeled (or equivalently, all the nodes have the same label). The terminal condition is that there is no non-empty connected (adjacent) bidomains. For example, there are still some adjacent bidomains, but for each bidomain $\langle \mathcal{V}'_{k1}, \mathcal{V}'_{k2} \rangle$, at least one of $\mathcal{V}'_{k1}$ and $\mathcal{V}'_{k2}$ is empty (containing no nodes), so there is no nodes to match in each bidomain. Examples are states 3 and 6 in Figure 1.
- Case 2: Nodes are labeled. For the terminal condition, there may be still some non-empty connected bidomains, but the node labels do not match causing no more node pairs to select from. For example, one bidomain contains $\mathcal{V}'_{k1}$ with C and N as node labels and $\mathcal{V}'_{k2}$ with H as node label. Then essentially there is no more node pairs left.

C.4. Promise-based Search: Improving Search with Backtracking

Unlike McSp or McSp+RL, GLSearch is optimized to find the largest common subgraph in one try, not to prune the search space. This is because, in practice, even with advanced pruning techniques, it is not practical to exhaust the entire search space for large graphs. As a consequence, GLSearch may still fall into local solutions if it follows the branch-and-bound algorithm. Thus, GLSearch improves upon McSp search by backtracking to an earlier state with the most promise of finding a larger subgraph when the current best solution is not improved upon within a fixed number of iterations. Typically, when the action space is larger, there is more potential for equally good or better actions to exist outside of the one selected, thus a state's action space size is equated to its promise.

In implementation, GLSearch keep track of promise by maintaining a priority queue of states, where a state's priority is given by its action space size, in parallel with the search stack. Thus, whenever GLSEARCH suspects the model is in a local solution, the next state popped on line 7 of Algorithm 1 in the main text will be the state with the largest action space from the priority queue, instead of the top state in the search stack. GLSEARCH detects when the model is in a local solution by keeping track of the largest subgraph found since the last time the priority queue was popped (or since the start of search). If this local best solution is not improved within a fixed number of iterations, the model knows it is in a local solution. In practice, we set this number to 3.

C.5. Notes on Equivalent States and Multiple Ground Truths

It is well known that for the MCS detection task, there can be multiple ground truth solutions with the same subgraph size. For example, in Figure 1, both states 3 and 6 correspond to the same subgraph size but the states 3 and 6 are different due to their different node-node mappings. Our model maintains the node-node mappings for each state, and therefore states 3 and 6 would be reached as different states. It is important to note that for large graph pairs, reaching both states 3 and 6 usually does not happen, since the search would first reach state 3 and need many iterations to backtrack to state 0 and further many iterations to reach state 6.

There is an even more subtle point in Figure 1. Suppose the search first matches node a to node 1, denoted as state 1, then matches node b to node 2, leading to state 2. After several iterations, it backtracks to state 0 and chooses to match node b to node 2, denoted as state x, then matches node a to node 1, denoted as state y. Although states 1 and x are different, but states 2 and y are equivalent, since both states 2 and y have the same node-node mapping, i.e. a to 1 and b to 2. Thus, the search maintains an additional set of visited states and at each time a new state is reached, a checking is performed to avoid revisiting the same state twice.

The node-node mapping is is an important component of the definition of state, not only because it differentiates the otherwise equivalent states, but also because different node-node mappings can lead to different final states and future reward (and thus must be considered by the design of DQN). Suppose the bidomain "01" in state 1 contains more than two nodes, i.e. there are many nodes connected to b in $\mathcal{G}_1$ besides node d and many nodes connected to node 2 in $\mathcal{G}_2$ besides node 3. Then state 2 is an intuitively more preferred state compared to state 5, since the matching of node b to node 2 allows more node pairs to be matched to each other in future, thus a larger action space in state 2. The value associated with state 2 thus should be larger than state 5.

C.6. Dealing with Large Action Space

For large graph pairs, the successful detection of MCS not only depend on the design of our critical DQN component as well as its training, but also rely on techniques which prune the large action space at each state.

The bidomain partitioning idea has been outlined in the main text which effectively reduces the action space size by only matching nodes in the same bidomain. However, for extremely large and dense graphs, the bidomains may not split the rest of the graphs enough and the action space may still be too large. For example, consider two fully connected graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, i.e. for every two nodes there is an edge. Then initially, there is no nodes selected, and there is only one bidomain consisting of all the node pairs. What is worse, at any state, there is always only one bidomain consisting of all the node pairs in the remaining subgraphs. Therefore, to reduce the action space further, we only compute the q scores for N_d nodes at most in each state. Specifically, we first sort the candidate bidomains by their size in ascending order, and select the first N_b small bidomains. Next, we sort the nodes in each bidomain by degree in descending order and select the first N_d/N_b nodes with large node degrees. For our experiments, $N_b = 1$ and $N_d = 20$ in GLSEARCH; $N_b = 1$ and $N_d = 3$ in GLSEARCH-SCAL. We find, in practice, these settings do not drastically alter performance.

In future, additional techniques for pruning the action space can be explored. For example, instead of bounding the computation to be N_d, we may perform a hierarchical graph matching by first running a clustering algorithm and then matching clusters which will be bounded by the number of clusters. Another possible direction is to learn an additional Q function learning which node is more promising instead of which node pair, i.e. $Q(s_t, a_t^{(i)})$ for node i and $Q(s_t, a_t^{(j)})$ for node j in the action $a_t = (i, j)$. We suppose such additional Q function may bring further performance gain.

C.7. Analysis of Time Complexity

Overall the branch-and-bound search has exponential worst-case time complexity due to the NP-hard nature of exact MCS detection, and our goal is to use additional overhead per search iteration to make "smarter" decision each iteration so that we can find a larger common subgraph faster (in less iterations *and* real running time). Per iteration, our model requires the neural network operations to compute a Q score instead of simply using a degree heuristic which is $\mathcal{O}(1)$. Here we analyze the time complexity of these neural operations:

- To compute the node embeddings, the complexity is the same as the GNN model, which in our case is $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$ for GAT (since nodes must aggregate embeddings from neighbors and attention scores must be

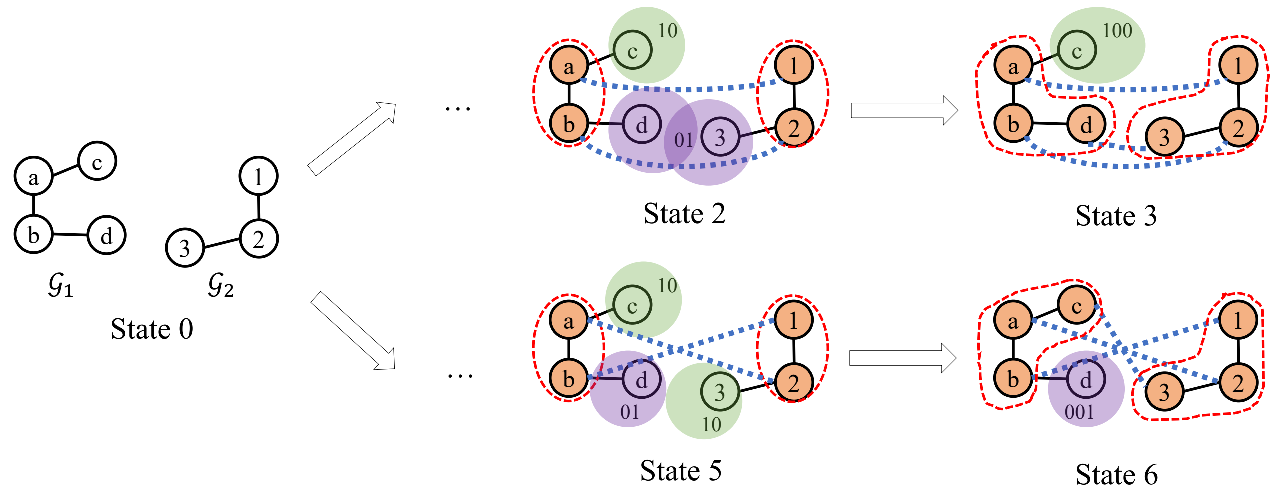


Figure 1: An example illustrating the idea of equivalent states. It is important to note that states 2 3, 5, 6 are different since their node-node mappings are different. However, the solutions derived from both states 3 and 6 have the same subgraph size, 3. In other words, there can be multiple ways to arrive at the same solution size, with different underlying sequential processes to reach the final states.

computed for each edge). Notice the node embeddings are computed by local neighborhood aggregation, and will not be updated in search, and therefore we compute the node embeddings only once at the beginning of search, and can be cached for efficiency.

- At each iteration, to compute a Q score for a state-action pair, we run Equation (3) (in the main text) which requires computing the whole-graph, subgraph, and bidomain embeddings. Overall the time complexity for each state-action pair is $\mathcal{O}(|\mathcal{V}| - |\mathcal{V}_s|)$ where $\mathcal{V}_s$ is the number of nodes in the currently matched subgraph. The whole-graph embeddings do not change across search, so they only need to be computed once at the beginning. The subgraph embeddings can be maintained incrementally, i.e. adding new node embeddings as search grows the subgraph. The bidomain embeddings are computed via a series of READOUT and INTERACT operations (Equation (2)): For READOUT: We use summation followed by MLP so the runtime is $\mathcal{O}(|\mathcal{V}| - |\mathcal{V}_s|)$; For INTERACT: We use a 1D CNN followed by MLP which depends on the embedding dimension set to a constant, and does not depend on the number of nodes in the input graphs.

Overall the time complexity for each iteration is $\mathcal{O}\big(N_d^2(|\mathcal{V}| - |\mathcal{V}_s|)\big)$.

D. Baseline Description and Comparison

For all the models used in experiments, we evaluate their performance under the same search framework, i.e. with consistent search iteration counting, upper bound estimation, etc. McSp and McSp+RL use heuristics to select node pairs, which is ineffective as shown in the main text and has been described in Section C.1. Therefore, this Section focuses on the comparison with the rest baselines, i.e. GW-QAP (Xu et al., 2019), I-PCA (Wang et al., 2019), and NeuralMcs (Bai et al., 2020).

GW-QAP performs Gromov-Wasserstein discrepancy (Peyré et al., 2016) based optimization *for each graph pair* and outputs a matching matrix $\boldsymbol{Y}$ for all node pairs indicating the likelihood of matching which is treated the same way as our q scores, i.e. at each search iteration we index into $\boldsymbol{Y}$ to select a node pair. I-PCA and NeuralMcs also output a matching matrix but require supervised training, and thus are trained using the same training data graph pairs as our GLSearch but with different loss functions and training signals. During testing, we apply the trained model on all testing graph pairs. For medium-size synthetic and real-world testing graph pairs, each method is given a budget of 500 search iterations. For large real-world graph pairs, each method is given a budget of 7500 search iterations. For million-node real-world graph pairs, each method is given a budget of 50 minutes. These budgets were chosen based on when the models' performances stabilized. We then describe each method in more details.

D.1. GW-QAP

GW-QAP is a state-of-the-art graph matching model for general graph matching. The task is not about MCS specifically, but instead about matching two graphs with its own criterion based on the Gromov-Wasserstein discrepancy (Peyré et al., 2016). Therefore, we suppose the matching matrix $\boldsymbol{Y}$ generated for each graph pair can be used as a guidance for which node pairs should be visited first. In other words, we pre-compute the matching scores for all the node pairs before the search starts, and in iteration, we look up the matching matrix and treat the score as the q score for action selection. I-PCA and NEURALMCS essentially compute a matching matrix too, and it is worth mentioning that all the three methods cannot learn a score based on both the state and the action. They can be regarded as generating the matching scores based on the whole graphs only without being conditioned and dynamically updated on states and actions.

D.2. I-PCA

I-PCA is a state-of-the-art image matching model, where each image is turned into a graph with techniques such as Delaunay triangulation (Lee & Schachter, 1980). It utilizes similarity scores and normalization to perform graph matching. We adapt the model to our task by replacing these layers with 3 GAT layers, consistent with GLSEARCH. As the loss is designed for matching image-derived graphs, we alter their loss functions to binary cross entropy loss similar to NEURALMCS which will be detailed below.

D.3. NEURALMCS

NEURALMCS is proposed for MCS detection with similar idea from I-PCA that a matching matrix is generated for each graph pair using GNNs. However, they both require the supervised training signals, i.e. the complete search for training graph pairs must be done to guide the update of I-PCA and NEURALMCS. In contrast, GLSEARCH is trained under the RL framework which does not require the complete search (in stage 2 and 3, only sequences are generated as detailed in Section C.2). This has the benefit of exploring the large action space in a “smarter” way and eventually allows our model to outperform I-PCA and NEURALMCS. In implementation, the complete search is not possible for large training graph pairs, so instead we apply a search budget and use the best solution found so far to guide the training of I-PCA and NEURALMCS.

Regarding the subgraph extraction strategy, for all the baselines, we use the same branch and bound algorithm, which is the state-of-the-art search designed for MCS (McCreesh et al., 2017). However, as mentioned in Section C.4, only our model is equipped with the ability to backtrack in a principled way. The main text shows the performance gain to GLSEARCH brought by the backtracking ability.

E. More Results with Analysis

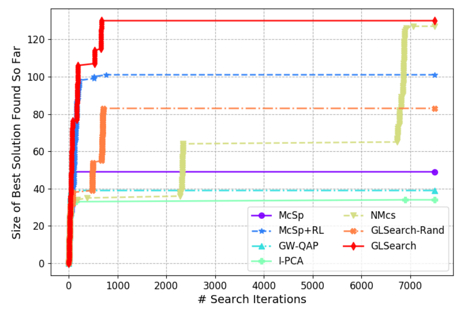
E.1. Best Solution Sizes across Time

Figure 2 shows that under the budget we set, for the large real-world graph pairs, all the methods reach a “steady state” where the best solution found so far no longer grows. This means the search continues but the search cannot find a larger solution, illustrating the fact that search gets “stuck”. In theory, given infinitely many iterations, all the models will eventually find the true MCS, which is the largest, but since the task is NP-hard, the search space can be exponential in the worst case (subject to pruning but in all the testing graph pairs the search has not finished yet), such budget has to be applied to search. At the end of the budget iterations, though, all the models have made “mistakes” (visiting unpromising states) and in order to find even larger common subgraphs, the search needs to backtrack potentially many times to fix those “mistakes” (backtracking to a very early state).

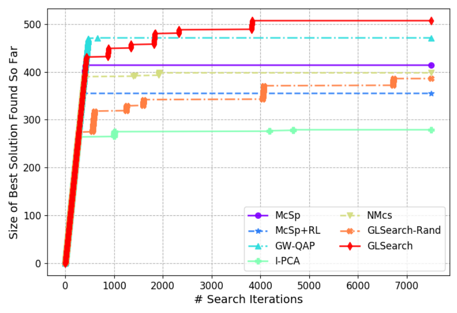
Admittedly, there is always the possibility that for more iterations, some baseline method may find a larger solution. Besides, in each iteration GLSEARCH does take more running time overhead as shown in Table 3. However, the point of our model is to quickly find a larger solution in as few iterations as possible, not to find a large solution given too many iterations. In other words, the goal of GLSEARCH is to be **smart** enough to **quickly** find a large solution instead of purely finding a good solution. Figure 2 shows that GLSEARCH not only finds solutions larger than baselines *when* the iteration budget is reached but also finds larger solutions faster than baselines *before* the budget is reached.

In addition to reaching a larger solution is less iterations, GLSEARCH also reaches better solutions with respect to runtime, as shown in Figure 3.[2] Notice, GLSEARCH finds the same large subgraph in 10 minutes as in 7500 iterations. Although per iteration, it is slower than MCSP per iteration, GLSEARCH finds a larger solution in usually less than a minute. Moreover, our implementation can be further optimized. Note, we adapted all baselines to run on Python for fair comparison, and made sure the search iteration counting is consistent across all baselines and the results shown in the main text. At this stage, our main goal is to explore the idea of ***“learning to search”***, which has been experimentally verified to be a promising direction of research, and leave the efforts of implementation optimization using various techniques as future focus.

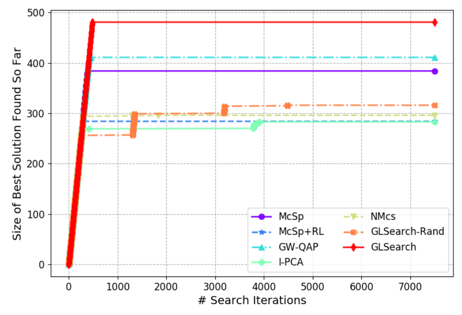
[2]In general, we refer to the running time results to show the performance gains of GLSEARCH and GLSEARCH-SCAL; however, to focus solely on the effects of different actions selected by different policies on the final common subgraph, we also compare by iterations.



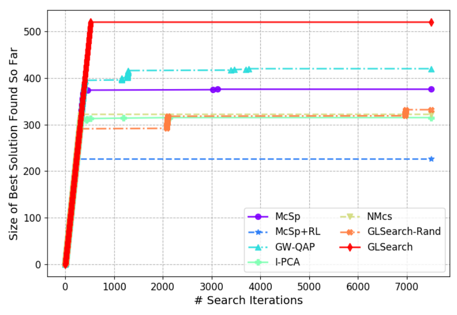
(a) Road



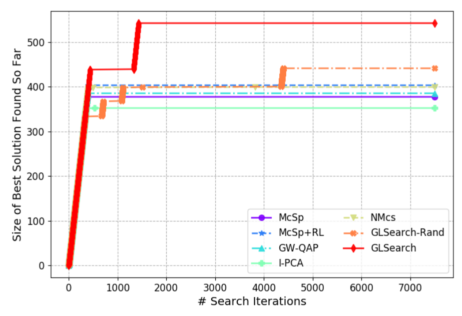
(b) DbEn



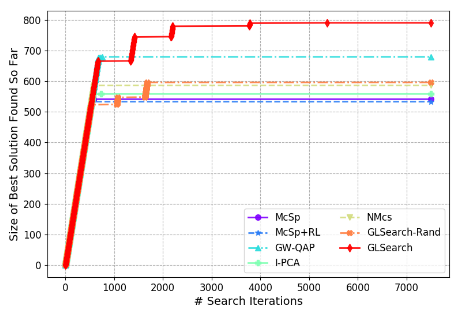
(c) DbZh



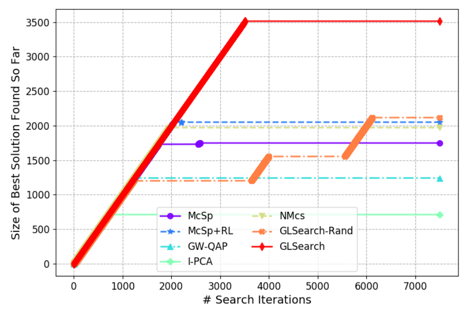
(d) DbPd



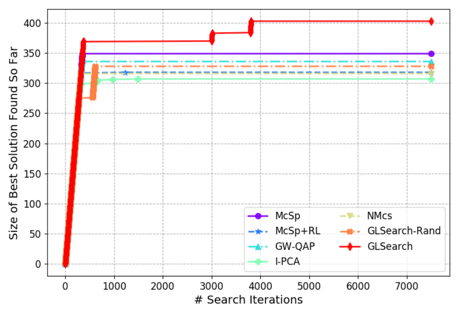
(e) Enro



(f) CoPr

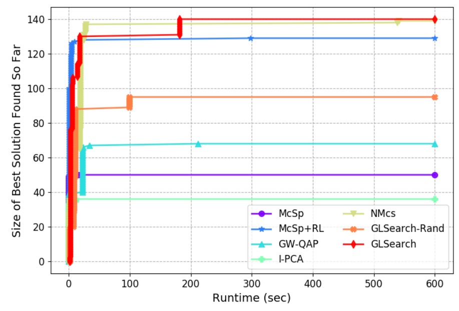


(g) Circ

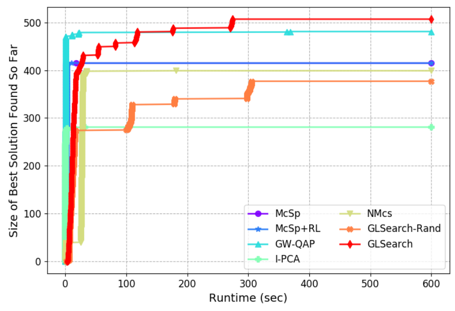


(h) HPpi

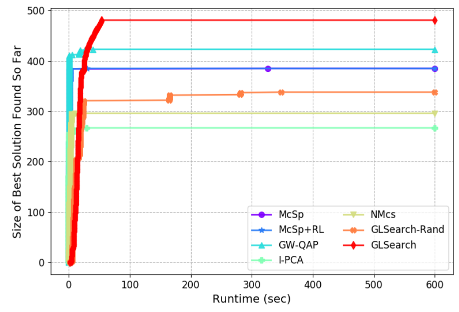
Figure 2: For each method, we maintain the best solution found so far in each iteration during the search process. We plot the size of the largest extracted common subgraphs found so far vs search iteration count for all the methods across all the datasets. The larger the subgraph size, the better ("smarter") the model in terms of quickly finding a large MCS solution under limited budget for large graphs.



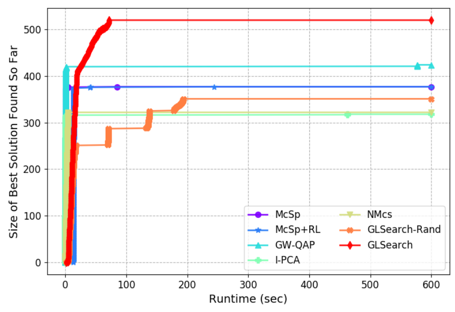
(a) ROAD



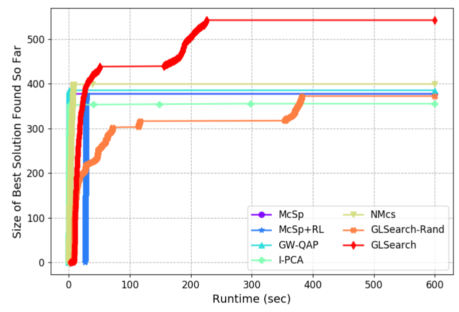
(b) DBEN



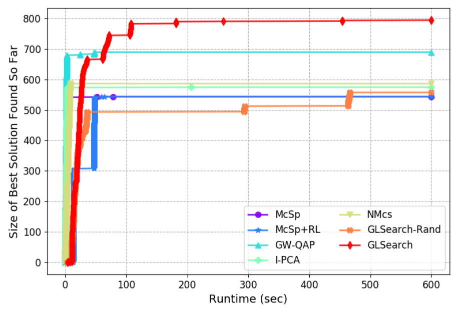
(c) DBZH



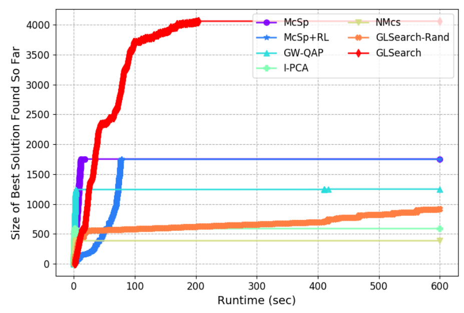
(d) DBPD



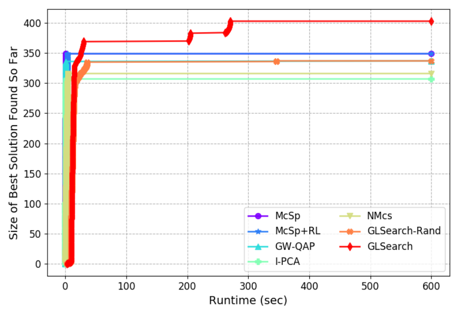
(e) ENRO



(f) COPR



(g) CIRC



(h) HPPI

Figure 3: For each method, we maintain the best solution found so far in each iteration during the search process. We plot the size of the largest extracted common subgraphs found so far vs the real running time for all the methods across all the datasets. The larger the subgraph size, the better ("smarter") the model in terms of quickly finding a large MCS solution under limited budget for large graphs.

Table 3: Average running time per iteration (msec).

Method	ROAD	DBEN	DBZH	DBPD	ENRO	COPR	CIRC	HPPI
MCSP	2.040	10.724	1.415	0.974	1.722	2.891	1.776	0.498
MCSP+RL	0.894	6.834	2.103	1.247	2.166	3.107	2.080	0.559
GW-QAP	0.548	0.834	0.546	4.692	1.041	3.419	1.550	0.546
I-PCA	1.152	1.797	0.967	0.897	1.739	2.725	3.792	0.636
NEURALMCS	2.394	4.172	4.648	5.667	9.610	9.788	7.471	15.742
GLSEARCH-RAND	17.392	66.418	67.342	67.946	163.005	71.972	655.447	83.488
GLSEARCH	8.132	66.552	71.409	96.262	135.087	51.181	37.377	60.509

E.2. Additional Ablation Study

Table 4 shows that pre-training and imitation learning benefit the performance under four out of the eight datasets. On ENRO and HPPI, without pre-training, our model performs better, which may be attributed to the fact that they are dense graphs (Table 1) while the training graphs used in stage 1 are relatively small and sparse (Section B.1).

Table 5 shows that the promised-based search improves the performance under four out of the eight datasets. For the other four datasets, the performance does not change, indicating that the backtracking to an earlier promising state based on the DQN output at least does not hurt the performance. In the cases like ENRO and HPPI, whose average node degrees are large, the promise-based search improves the performance by a large amount, showing the usefulness of the proposed strategy.

E.3. Results on Graph Pairs with Known MCS Size Lower Bound

To better understand the quality of subgraphs found by GLSEARCH, we construct new datasets with known lower bound MCS sizes. This is accomplished through generating 2 new graphs that share a common subgraph from the existing large real-world graph datasets. For each real-world graph, $\mathcal{G}_0$, we randomly extract 3 different subgraphs of the same size, $\mathcal{S}_0$, $\mathcal{S}_1$, and $\mathcal{S}_2$, by running breadth first search from 3 different starting nodes and extracting the explored induced subgraph. To construct a new graph pair, $(\mathcal{G}_1, \mathcal{G}_2)$, we form $\mathcal{G}_1$ by connecting $\mathcal{S}_0$ to $\mathcal{S}_1$ with 20 random edges and form $\mathcal{G}_2$ by connecting $\mathcal{S}_0$ to $\mathcal{S}_2$ with 20 random edges. Thus, connections between $\mathcal{S}_0$ nodes are the same in both $\mathcal{G}_1$ and $\mathcal{G}_2$, but connections between $\mathcal{G}_1 \setminus \mathcal{S}_0$ and $\mathcal{G}_2 \setminus \mathcal{S}_0$ are different. Notice, the lower bound of MCS size in these new datasets would be $|\mathcal{S}_0|$, and we name the new dataset by adding 'ss' to the parent dataset's name.

The results on CIRC and these datasets (Table 6) show no MCS method guarantees to always detect a solution that is as large as the known MCS solution/lower bound. This suggests the difficulty of the task itself. In practice, though, GLSEARCH is still preferred compared to baselines due to its better performance.

E.4. Result Visualization on ROAD-CA and ROAD-TX

For the largest two graph pairs, ROAD-CA and ROAD-TX, in order to clearly see and compare the subgraph growth across time of GLSEARCH-SCAL and MCSP, we perform the following visualization: At every 1000 or 2000 iterations, we plot the graph pair and highlight the matched subgraphs. As shown in Figures 4 and 5, from the left to the right, the growing of the extracted common subgraphs can be seen.

In order to render the best visualization, the following techniques are used: (1) Since the input graphs are two large, we only plot the extracted subgraphs and the remaining graphs around the extracted subgraphs by performing breadth-first search starting from the extracted subgraphs (gray color); (2) For the matched subgraphs, to clearly see the node-node mapping, we ensure the node layout positions are the same across $\mathcal{G}_1$ (top) and $\mathcal{G}_2$ (bottom), and use colors for the matched subgraph nodes to indicate the node-node mapping[3].

It is noteworthy that on ROAD-TX, MCSP only finds 30 nodes as shown in Figure 5. By examining the plot carefully, we can see that this is caused by poor choice of actions that make further selections of node pairs impossible, i.e. there is no more node pairs to choose in action space, reaching a terminal condition[4]. As shown in Figure 4 in the main text, MCSP spends the rest of the time backtracking and exploring the rest of the search space, but given the exponentially growing search space size, MCSP cannot easily leave the local optimum. Given infinitely long running time, however, all methods under the branch and bound algorithm will eventually find the exact MCS solution, which is an impractical assumption in real world. This illustrates the necessity of making smart node pair selection choices through the search instead of relying on heuristics. As a fact, MCSP+RL also finds 30 nodes, since it degenerates

[3]The nodes of ROAD-CA and ROAD-TX are unlabeled and the colors are only used to highlight node-node mapping.

[4]We aim to find induced common subgraphs, meaning that if a new node pair is selected, all the edges between the new nodes and the existing nodes must also be included, and in this example, any new node pair would lead to the resulting subgraphs **not** isomorphic to each other.

Table 4: Contribution of pre-training and imitation learning to the performance of GLSEARCH. "no-sup" denotes the removal of the pre-training stage (The first 3750 iterations: IL; The last 6250 iterations: Normal QDN training); "no IL" denotes the removal of the imitation learning stage (The first 3750 iterations: pre-training; The last 6250 iterations: normal DQN training); "no sup; no IL" indicates the entire training (10000 iterations) is normal DQN training.

Method	ROAD	DBEN	DBZH	DBPD	ENRO	COPR	CIRC	HPPI
GLSEARCH (no sup)	0.557	0.957	0.946	0.904	**1.000**	0.999	**1.000**	**1.000**
GLSEARCH (no IL)	**1.000**	0.933	0.965	0.887	0.357	0.875	0.666	0.632
GLSEARCH (no sup; no IL)	0.678	0.907	0.896	0.837	0.401	0.949	0.949	0.651
GLSEARCH	0.879	**1.000**	**1.000**	**1.000**	0.412	**1.000**	0.855	0.688
BEST SOLUTION SIZE	149	486	465	471	1318	790	4112	587

Table 5: Contribution of promise-based search (Section C.4) to the performance of GLSEARCH. "no promise" denotes that the search does not use the proposed promise-based search, i.e. it does not backtrack to an earlier state if the search makes no progress after a certain amount of iterations, and instead, it continues the regular branch and bound search.

Method	ROAD	DBEN	DBZH	DBPD	ENRO	COPR	CIRC	HPPI
GLSEARCH (no promise)	0.879	**1.000**	**1.000**	**1.000**	0.412	**1.000**	0.855	0.688
GLSEARCH	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
BEST SOLUTION SIZE	131	508	482	521	543	791	3515	404

to the same policy as MCSP at the beginning of each new graph pair as mentioned in the main text. We verify that its found subgraphs are indeed exactly the same as MCSP.

Another insight from Figure 5 is that, the initial node pair selection made by MCSP, which is highlighted as "nodes with high degrees" in the plot, misleads MCSP into eventually finding a very small solution. In contrast, GLSEARCH-SCAL uses embeddings at subgraph, whole-graph, and bidomain levels to make a decision at each step, which capture the network structure better than just the node degree information used by MCSP. A real-world analogy can be road networks in downtown areas which tend to be grid structures versus road networks in rural areas which tend to be less rigid. GLSEARCH-SCAL takes graph structure into account and matches one downtown area with another downtown area, so the resulting matched subgraphs can be very large, while MCSP is misled by its heuristic into matching two high-degree nodes[5] at the beginning of search, but unfortunately the two areas do not have similar road network structures and eventually the matched subgraphs are small. It must be noted, however, that this analogy is only a high-level hypothesis for this specific case, and in general, for more complicated graph structures, the actual decisions made by GLSEARCH-SCAL usually cannot be easily explained using such an analogy.

F. Extensions of GLSEARCH

GLSEARCH can be extended for a flurry of other MCS definitions, e.g. approximate MCS, MCS for weighted and directed graphs, etc. via a moderate amount of change to the search and learning components. In this section, we briefly outline what could be done for these tasks.

For approximate MCS detection, the bidomain constraint must be relaxed. One method of relaxing this constraint is to allow sets of nodes belonging to different but similar bidomains to match to each other. For instance, nodes in $\mathcal{G}_1$ from the bidomain of bitstring "00110" could map with nodes in $\mathcal{G}_2$ from the bidomain of bitstring "00111", since they are only 1 hamming distance away. Such relaxations as this can be made stricter or looser based on the application. The difference would be the search framework, thus the learning part of GLSEARCH can largely stay the same.

Regarding MCS for graphs with non-negative edge weights, assuming our task is to maximize the sum of edge weights in the MCS, instead of defining $r_t = 1$, we can alter the reward function to be the difference of the sum of edge weights before and after selecting a node pair $r_t = \Sigma_{e \in S_t^{(u,v)}} w(e) - \Sigma_{e \in S_t} w(e)$ where S_t is the edges of currently selected subgraph, $S_t^{(u,v)}$ is the edges of the subgraph after adding node pair, (u, v), and $w(\cdot)$ is a function that takes and edge and returns its weight. As the cumulative sum of rewards at step T is the sum of edge weights $\Sigma_{t \in [1,\dots,T]} r_t = \Sigma_{e \in S_T} w(e)$ and reinforcement learning aims to maximize the cumulative sum of rewards, we can adapt GLSEARCH to optimize for MCS problems with weighted edges.

Regarding MCS for directed graphs, the bidomain constraint may be altered such that every bit in the bidomain string representations now has 3 states: '0' for disconnected, '1' for connected by in-edge, and '2' for connected by out-edge.

[5]The node degree of both nodes is 12 and we verify they are the highest-degree nodes in $\mathcal{G}_1$ and $\mathcal{G}_2$.

Table 6: Results on graph pairs with a common core subgraph (lower bound of MCS), with a fixed runtime of 10 minutes.

Method	**ROAD-SS** 444	**DBEN-SS** 778	**DBZH-SS** 762	**ENRO-SS** 1346	**COPR-SS** 1406	**HPPI-SS** 860
MCSP	0.588	0.466	0.544	0.216	1.000	0.233
MCSP+RL	0.588	0.466	0.544	0.214	1.000	0.233
GLSEARCH	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
BEST SOLUTION SIZE	188	389	350	673	703	430
CORE (LOWER BOUND) SIZE	222	389	381	673	703	430

By considering the inward/outward direction of a bitstring, we can guarantee the isomorphism of directed graphs. In this case, the search framework would only differ in how bidomains are partitioned. The learning part of GLSEARCH would stay the same for this application.

Regarding MCS for chemical compounds, we are aware of works that aim to tackle MCS in the chemoinformatics domain (Schietgat et al., 2013; Duesbury et al., 2018), and we do admit that the current definition of MCS may not satisfy constraints in the chemoinformatics domain, e.g. Figure 6. However, since we aim to design a general solver, we believe our current work has strong potential to be extended in the future with domain-specific constraints.

More generally, we believe that there are many more extensions to GLSEARCH in addition to the ones listed, such as disconnected MCS, network alignment, or subgraph extraction. Further exploration of these are to be done as future efforts.

G. Additional Result Visualization

We plot the testing graph pairs and the results of MCSP and GLSEARCH in this Section using a software called Gephi (Bastian et al., 2009). For all the figures except Figure 19, we use two colors for nodes, one for the selected subgraphs by the model, the other for the remaining subgraphs that cannot be further matched within the search budget. When plotting, we use larger circle size for nodes with larger degrees.

In general, GLSEARCH is a less interpretable but more powerful method compared to heuristic baselines. That said, GLSEARCH presents some insights that may be useful for producing new hand-crafted heuristics.

GLSEARCH identifies "smart" nodes which can lead to larger common subgraphs faster. For example, in the road networks (ROAD), as in Figure 7 and 8, our learned policy selects nodes with smaller degrees which allow for easier matching. The common subgraphs in road networks are most likely long chains, where nodes tend to have low degrees. In contrast MCSP always chooses high-degree nodes first leading to smaller extracted subgraphs.

GLSEARCH identifies "smart" *matching* of nodes which can lead to larger common subgraphs faster. For example, in the circuit graph (CIRC), we find 3 high-degree nodes that, when correctly matched, greatly reduces the matching difficulty of remaining nodes (see Figure 19 and 20). Upon further analysis, MCSP incorrectly matches the 3 high degree nodes (matching high degree node to low degree node). This happens when matching high-degree node correctly would break the isomorphism constraint (due to the current selected subgraph being incorrectly matched). GLSEARCH conscientiously adds node pairs so that it will always be able to match the 3 high degree nodes correctly.

We believe two aspects of GLSEARCH design lead to this phenomenon. First, GLSEARCH encodes neighborhood structures that are k-hop away. MCSP only looks at a single node and not its relationship with k-hop neighbors. Second, GLSEARCH considers scores on the node-node pair granularity, thus it will only match nodes with similar local neighborhoods. MCSP only considers scores on the node granularity, potentially matching 2 nodes with dissimilar neighborhoods together.

From these insights, one can potentially design a heuristic to first detect highly valuable nodes and guide a policy which prioritizes the matching of these critical nodes, or create better heuristics that consider not only uses the features of a single node but also the similarity between the 2 nodes being matched.

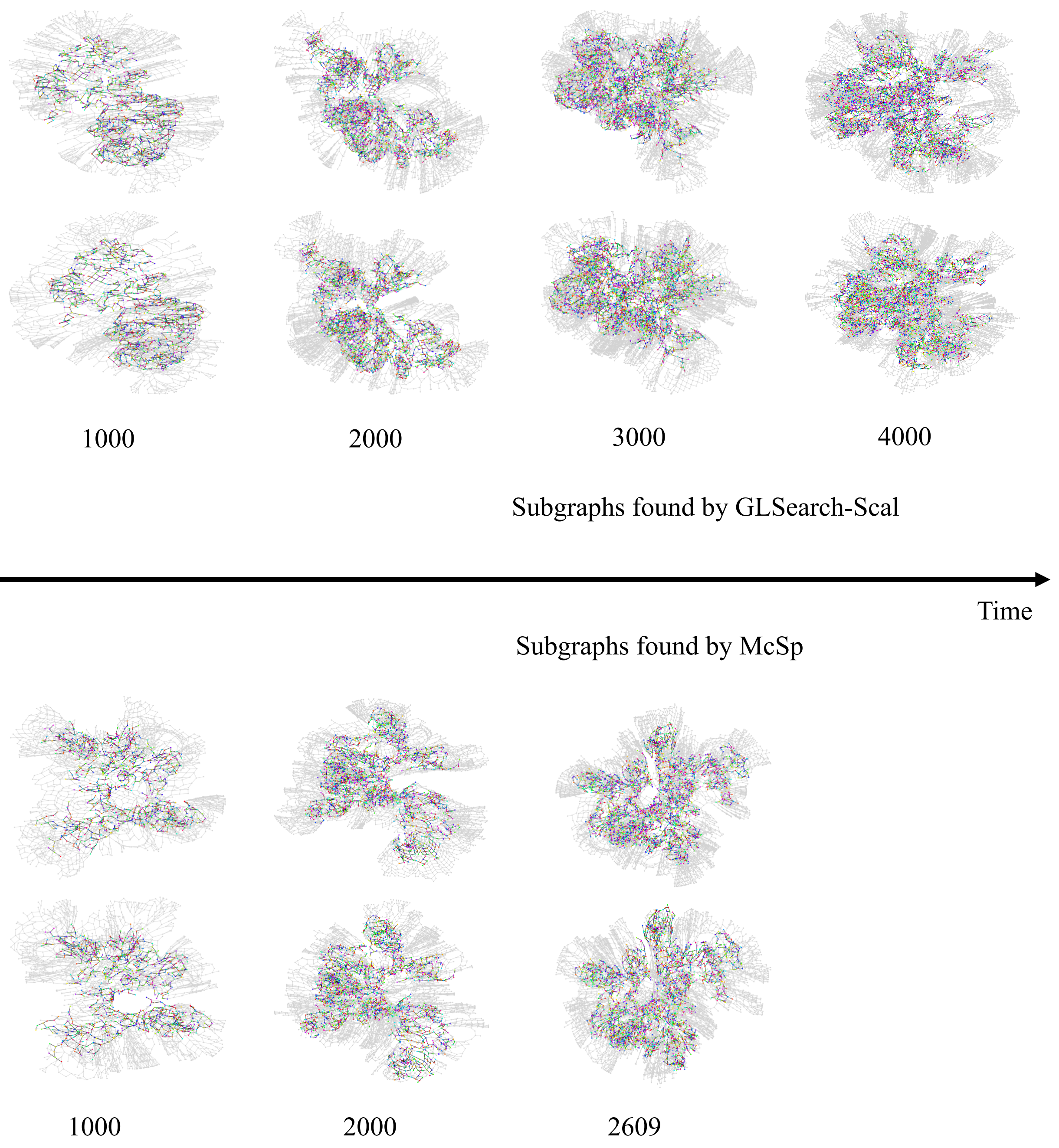


Figure 4: Visualization of subgraphs found by GLSEARCH-SCAL and MCSP on ROAD-CA. The subgraphs found by each method grows across time (until the budget of 50 minutes is reached), and the sizes of the subgraphs are denoted at the bottom of each figure. MCSP only finds 2609 nodes as shown in Figure 4 (a) in the main text, and the solution does not increase further.

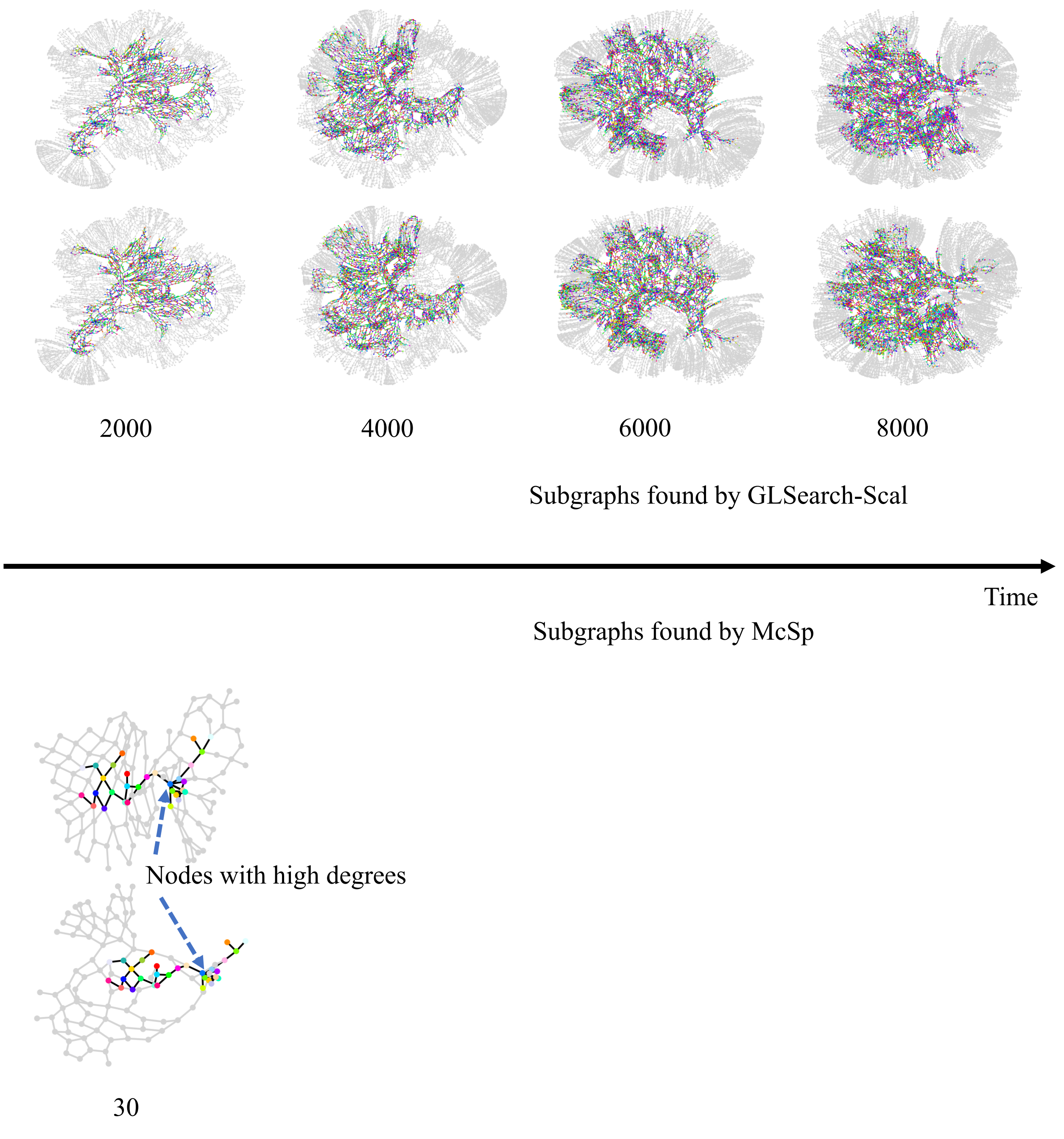


Figure 5: Visualization of subgraphs found by GLSEARCH-SCAL and MCSP on ROAD-TX. The subgraphs found by each method grows across time (until the budget of 50 minutes is reached), and the sizes of the subgraphs are denoted at the bottom of each figure. MCSP only finds 30 nodes as shown in Figure 4 (b) in the main text, and the solution does not increase further.

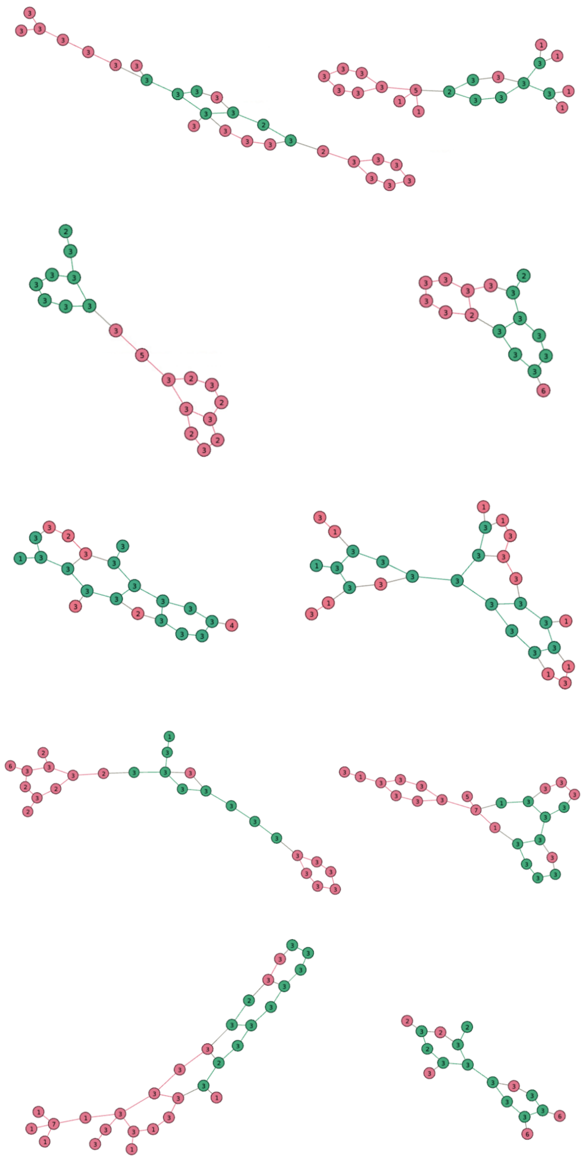


Figure 6: Visualization of 5 sampled graph pairs with the MCS results by GLSEARCH on NCI109. Each chemical compound node has its label indicated in the plot. Extracted subgraphs are highlighted in green.

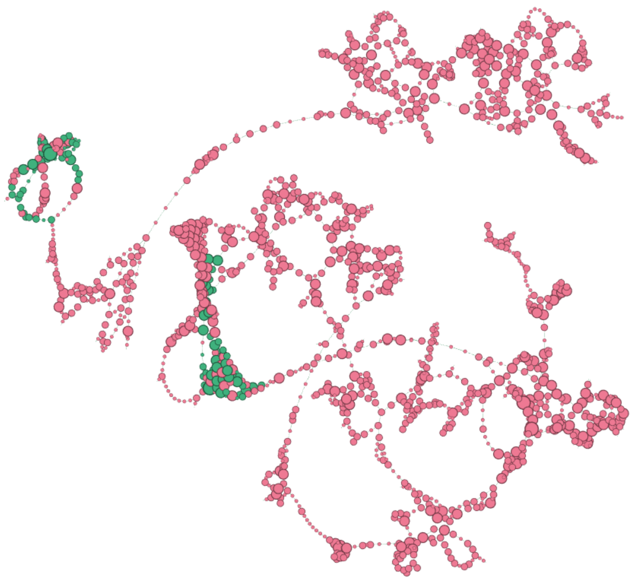


Figure 7: Visualization of MCSP result on ROAD. Extracted subgraphs are highlighted in green.

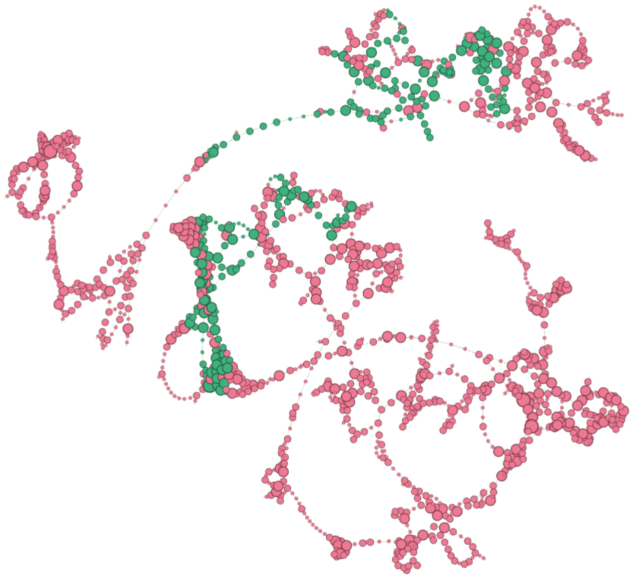


Figure 8: Visualization of GLSEARCH result on ROAD. Extracted subgraphs are highlighted in green.

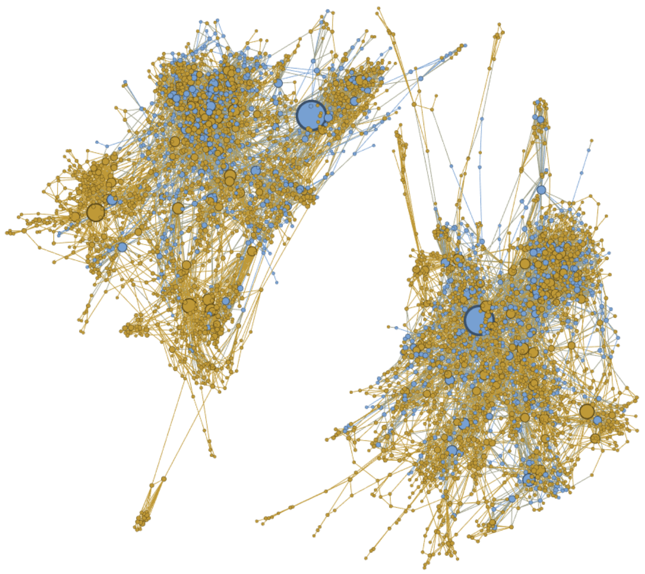


Figure 9: Visualization of McSp result on DbEn. Extracted subgraphs are highlighted in blue.

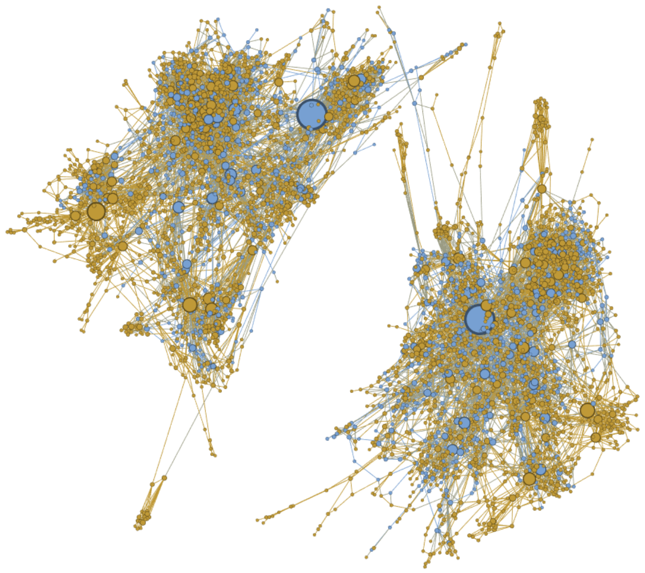


Figure 10: Visualization of GLSearch result on DbEn. Extracted subgraphs are highlighted in blue.

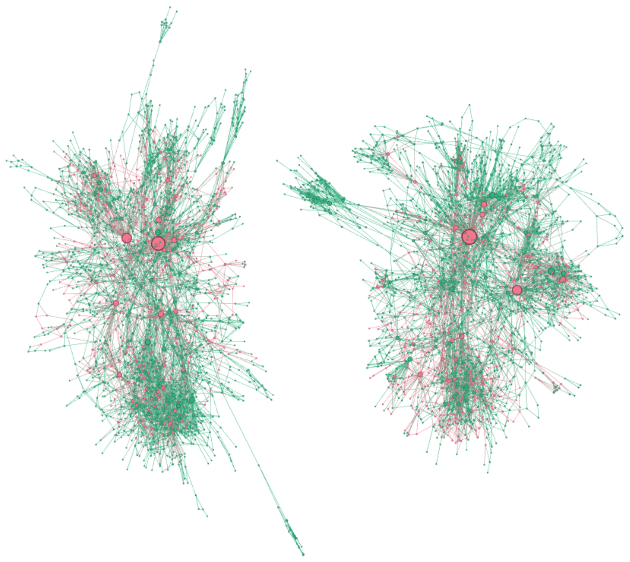


Figure 11: Visualization of McSp result on DbZh. Extracted subgraphs are highlighted in pink.

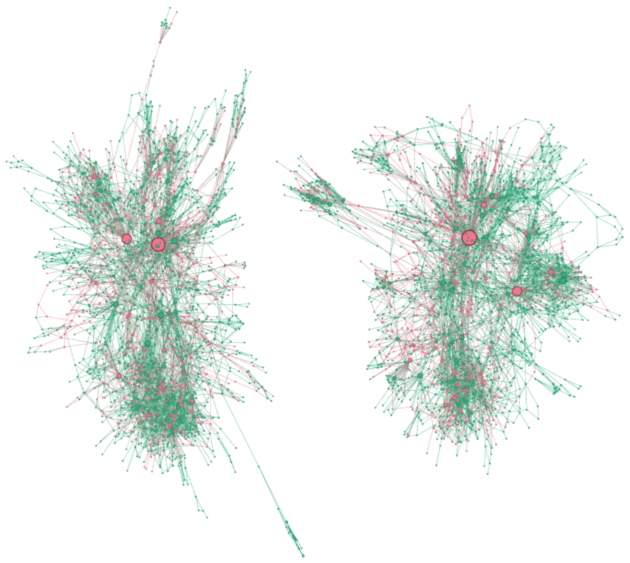


Figure 12: Visualization of GLSearch result on DbZh. Extracted subgraphs are highlighted in pink.

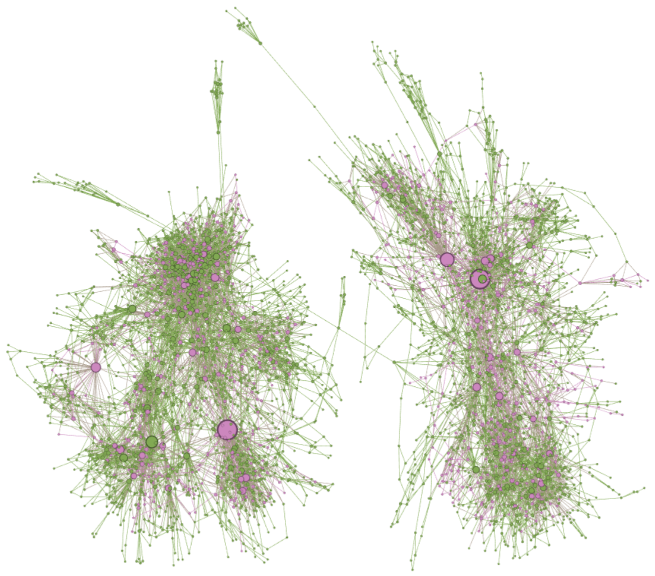


Figure 13: Visualization of MCSP result on DBPD. Extracted subgraphs are highlighted in purple.

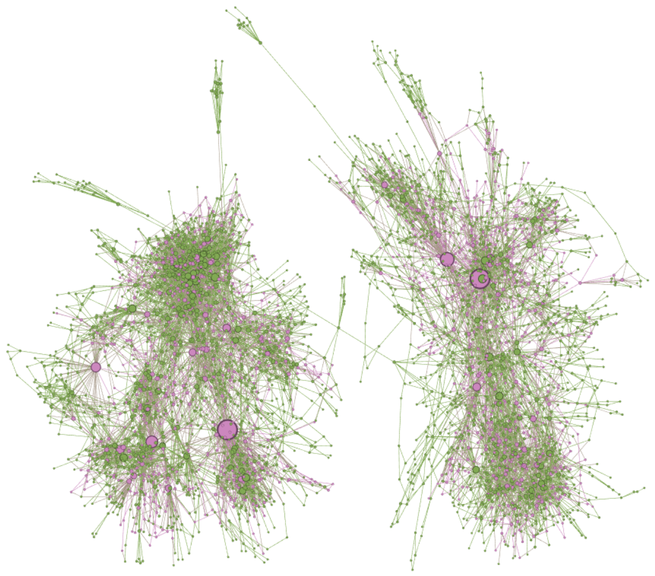


Figure 14: Visualization of GLSEARCH result on DBPD. Extracted subgraphs are highlighted in purple.

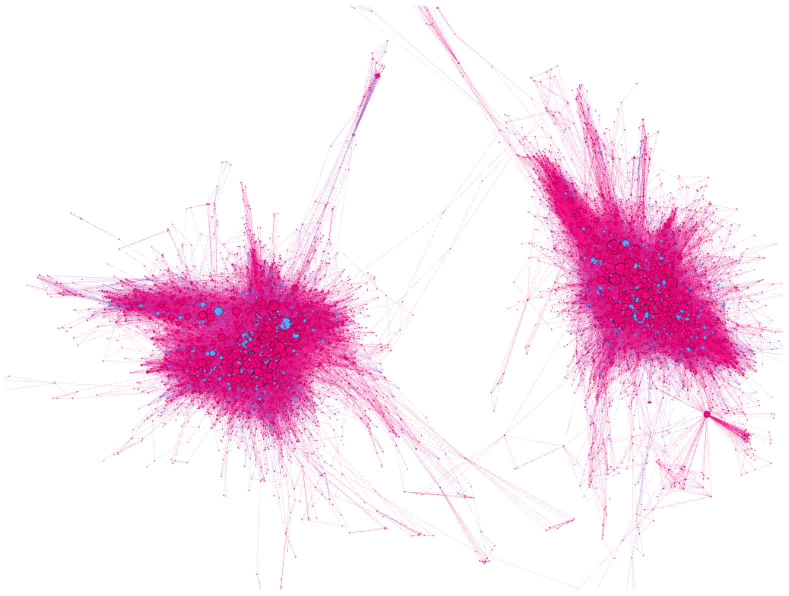


Figure 15: Visualization of McSp result on Enro. Extracted subgraphs are highlighted in blue.

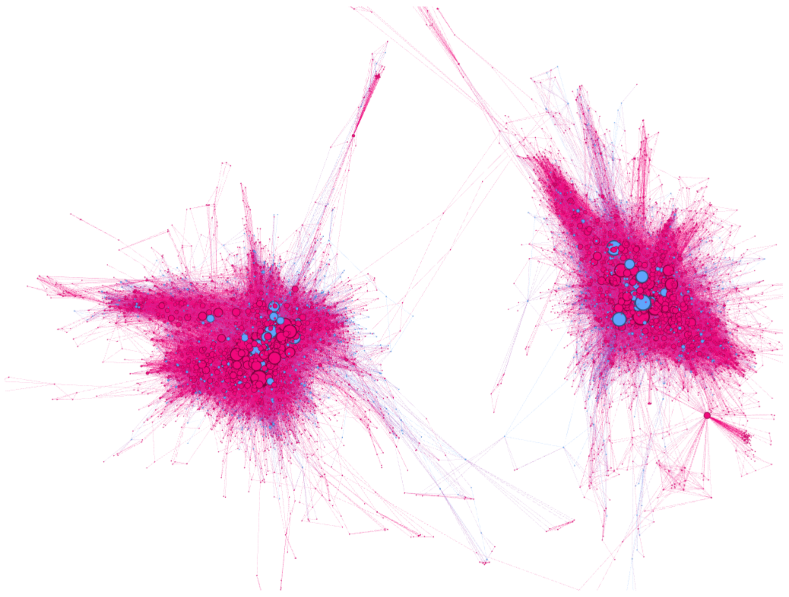


Figure 16: Visualization of GLSearch result on Enro. Extracted subgraphs are highlighted in blue.

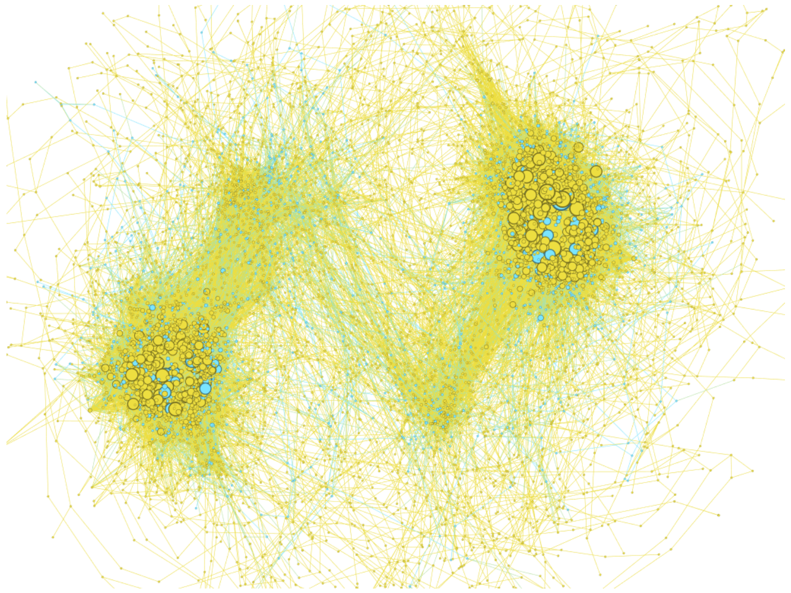


Figure 17: Visualization of McSp result on CoPr. Extracted subgraphs are highlighted in blue.

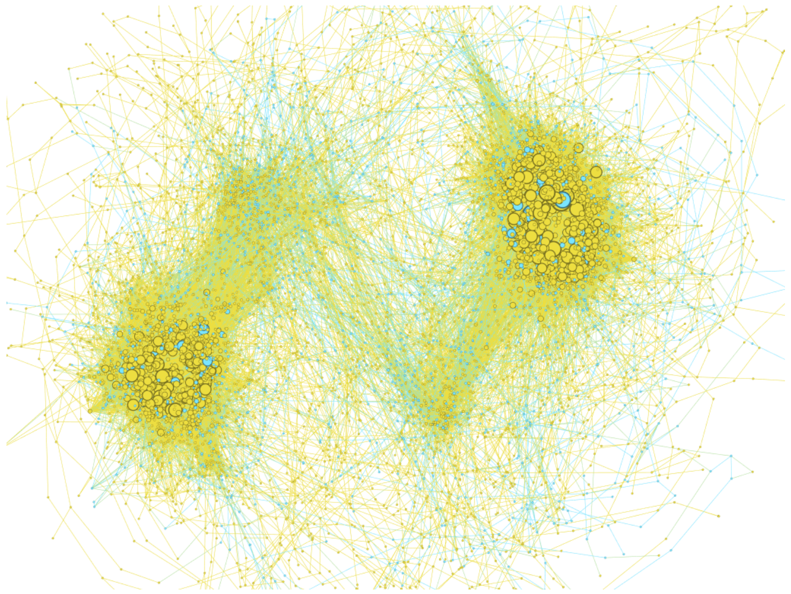


Figure 18: Visualization of GLSearch result on CoPr. Extracted subgraphs are highlighted in blue.

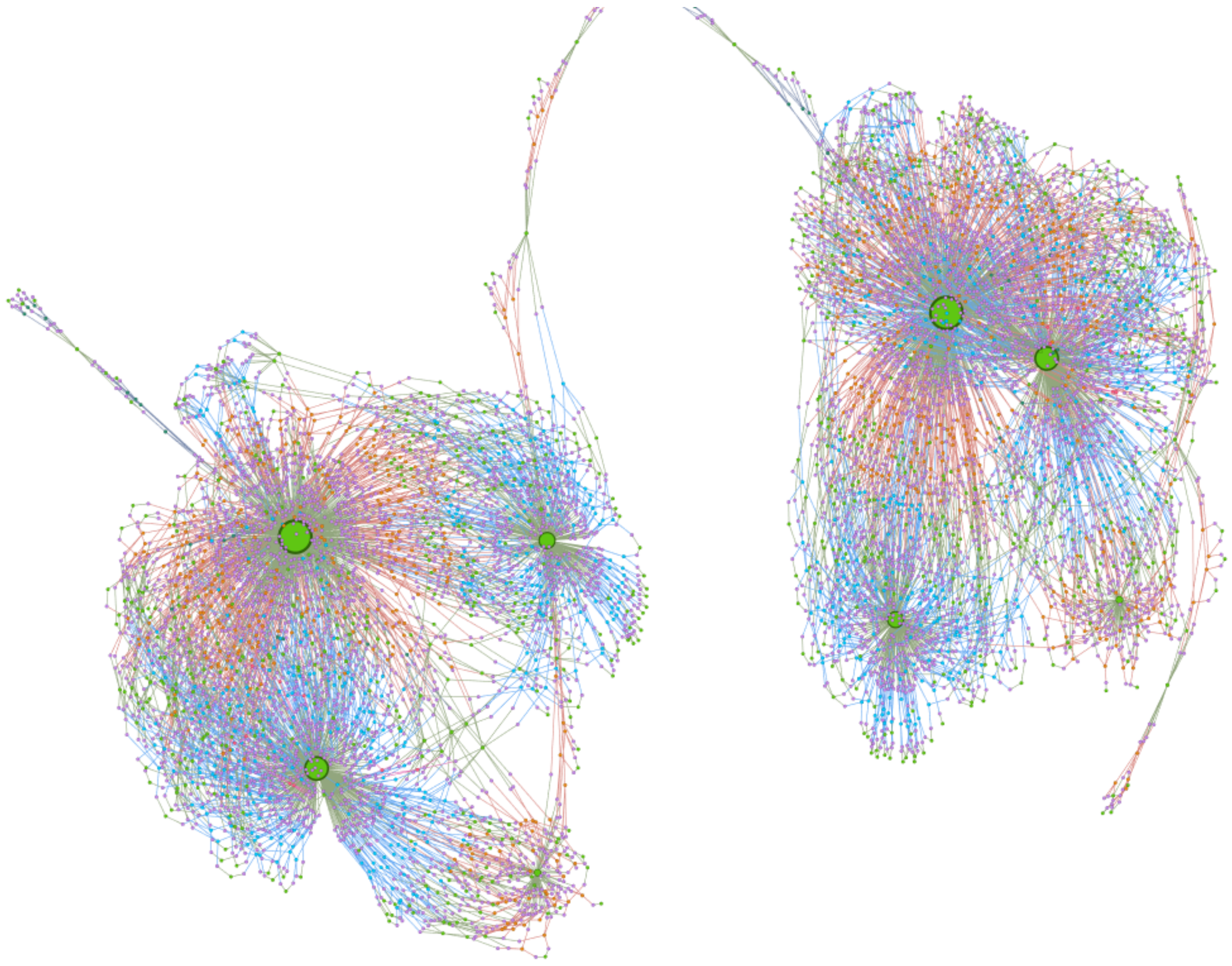


Figure 19: Visualization of the original graph pair of CIRC. The two graphs are in fact isomorphic. Different colors denote different node labels. There are 6 node labels in total: M (71.67%), null (10.41%), PY (9.1%), NY (8.23%), N (0.37%), and P (0.21%).

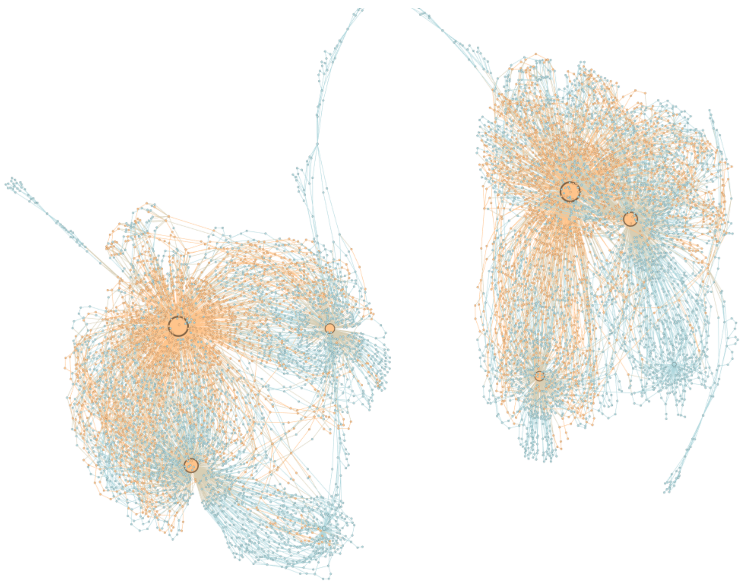


Figure 20: Visualization of McSp result on Circ. Extracted subgraphs are highlighted in yellow.

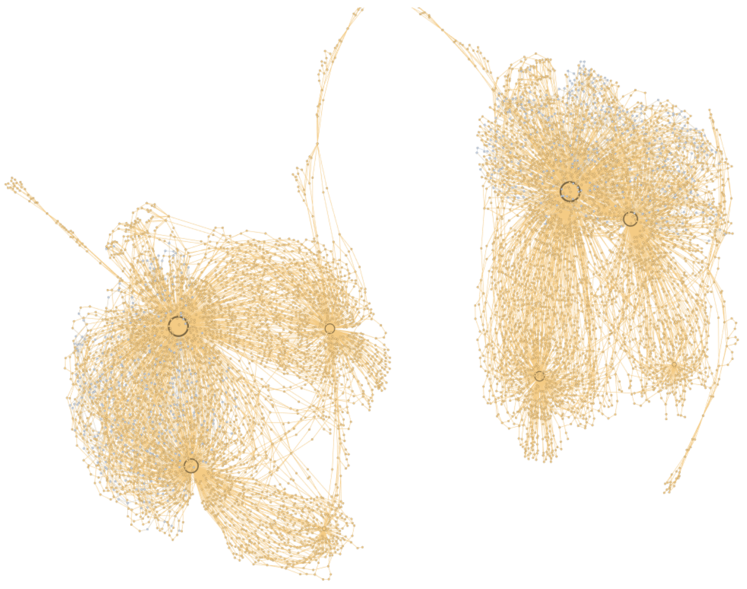


Figure 21: Visualization of GLSearch result on Circ. Extracted subgraphs are highlighted in yellow.

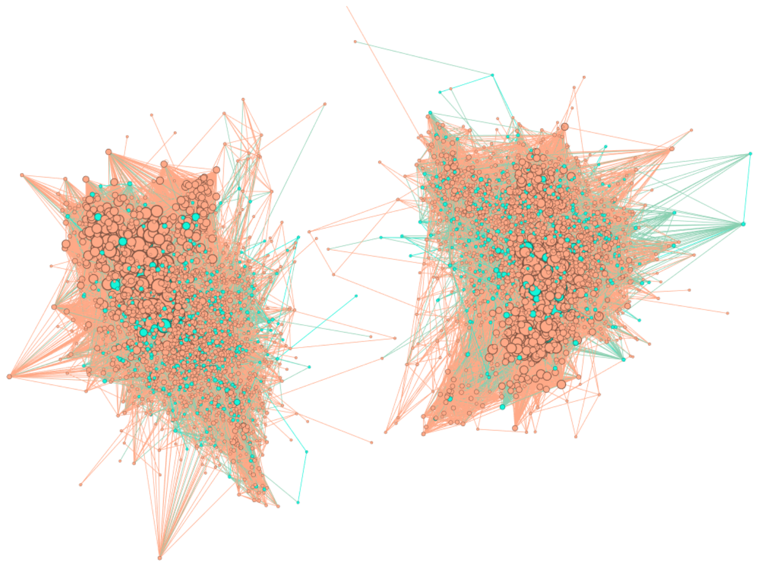


Figure 22: Visualization of McSp result on HPpi. Extracted subgraphs are highlighted in cyan.

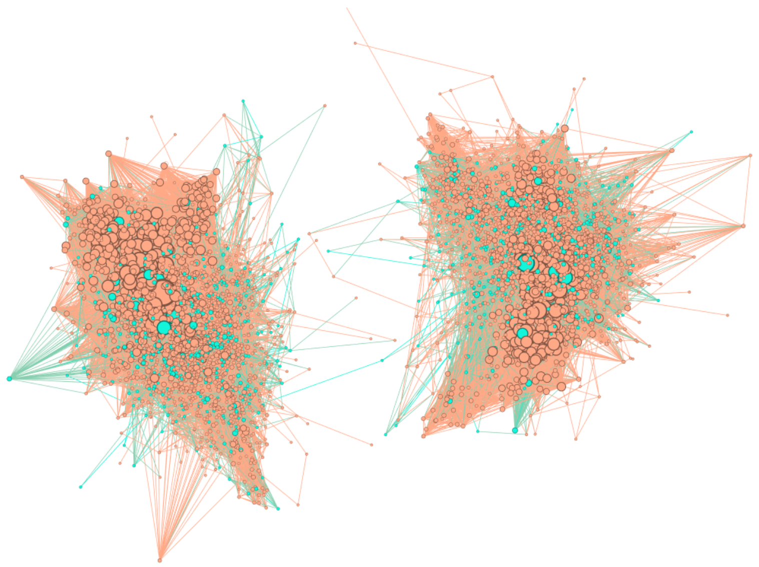


Figure 23: Visualization of GLSearch result on HPpi. Extracted subgraphs are highlighted in cyan.

References

Agrawal, M., Zitnik, M., Leskovec, J., et al. Large-scale analysis of disease pathways in the human interactome. In *PSB*, pp. 111–122. World Scientific, 2018.

Bai, Y., Ding, H., Bian, S., Chen, T., Sun, Y., and Wang, W. Simgnn: A neural network approach to fast graph similarity computation. *WSDM*, 2019.

Bai, Y., Xu, D., Gu, K., Wu, X., Marinovic, A., Ro, C., Sun, Y., and Wang, W. Neural maximum common subgraph detection with guided subgraph extraction, 2020. URL https://openreview.net/forum?id=BJgcwh4FwS.

Barabási, A.-L. and Albert, R. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

Bastian, M., Heymann, S., and Jacomy, M. Gephi: an open source software for exploring and manipulating networks. In *Proceedings of the International AAAI Conference on Web and Social Media*, volume 3, 2009.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. Curriculum learning. In *ICML*, pp. 41–48, 2009.

Debnath, A. K., Lopez de Compadre, R. L., Debnath, G., Shusterman, A. J., and Hansch, C. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of medicinal chemistry*, 34 (2):786–797, 1991.

Duesbury, E., Holliday, J., and Willett, P. Comparison of maximum common subgraph isomorphism algorithms for the alignment of 2d chemical structures. *ChemMedChem*, 13(6):588–598, 2018.

Gilbert, E. N. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

Klimt, B. and Yang, Y. Introducing the enron corpus. In *CEAS*, 2004.

Lee, D.-T. and Schachter, B. J. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

Leskovec, J., Lang, K. J., Dasgupta, A., and Mahoney, M. W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

McCreesh, C., Prosser, P., and Trimble, J. A partitioning algorithm for maximum common subgraph problems. 2017.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *NeurIPS Deep Learning Workshop 2013*, 2013.

Peyré, G., Cuturi, M., and Solomon, J. Gromov-wasserstein averaging of kernel and distance matrices. In *ICML*, pp. 2664–2672, 2016.

Riesen, K. and Bunke, H. Iam graph database repository for graph based pattern recognition and machine learning. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pp. 287–297. Springer, 2008.

Schietgat, L., Ramon, J., and Bruynooghe, M. A polynomial-time maximum common subgraph algorithm for outerplanar graphs and its application to chemoinformatics. *Annals of Mathematics and Artificial Intelligence*, 69(4):343–376, 2013.

Shchur, O., Mumme, M., Bojchevski, A., and Günnemann, S. Pitfalls of graph neural network evaluation. *Relational Representation Learning Workshop (R2L 2018), NeurIPS 2018*, 2018.

Shrivastava, A. and Li, P. A new space for comparing graphs. In *Proceedings of the 2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pp. 62–71. IEEE Press, 2014.

Sun, Z., Hu, W., and Li, C. Cross-lingual entity alignment via joint attribute-preserving embedding. In *International Semantic Web Conference*, pp. 628–644. Springer, 2017.

Wale, N., Watson, I. A., and Karypis, G. Comparison of descriptor spaces for chemical compound retrieval and classification. *Knowledge and Information Systems*, 14 (3):347–375, 2008.

Wang, R., Yan, J., and Yang, X. Learning combinatorial embedding networks for deep graph matching. *ICCV*, 2019.

Wang, X., Ding, X., Tung, A. K., Ying, S., and Jin, H. An efficient graph indexing method. In *ICDE*, pp. 210–221. IEEE, 2012.

Watts, D. J. and Strogatz, S. H. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440, 1998.

Xu, H., Luo, D., and Carin, L. Scalable gromov-wasserstein learning for graph partitioning and matching. In *NeurIPS*, pp. 3046–3056, 2019.

Yanardag, P. and Vishwanathan, S. Deep graph kernels. In *SIGKDD*, pp. 1365–1374. ACM, 2015.

Zeng, Z., Tung, A. K., Wang, J., Feng, J., and Zhou, L. Comparing stars: On approximating graph edit distance. *PVLDB*, 2(1):25–36, 2009.