Detecting Small Query Graphs in A Large Graph via Neural Bi-Level Search

Anonymous Author(s)
Affiliation
Address
email

Abstract

Recent advances have shown the success of using reinforcement learning and search to solve NP-hard graph-related tasks, such as Traveling Salesman Optimization, Graph Edit Distance computation, etc. However, it remains unclear how one can efficiently and accurately detect the occurrences of a small query graph in a large target graph, which is a core operation in graph database search, biomedical analysis, social group finding, etc. This task is called Subgraph Matching which essentially performs subgraph isomorphism check between a query graph and a large target graph. One promising approach to this classical problem is the "learning-to-search" paradigm, where a reinforcement learning (RL) agent is designed with a learned policy to guide a search algorithm to quickly find the solution without any solved instances for supervision. However, for the specific task of Subgraph Matching, though the query graph is usually small given by the user as input, the target graph is often orders-of-magnitude larger. It poses challenges to the neural network design and can lead to solution and reward sparsity. In this paper, we propose N-BLS with two innovations to tackle the challenges: (1) A novel encoder-decoder neural network architecture to dynamically compute the matching information between the query and the target graphs at each search state; (2) A Monte Carlo Tree Search enhanced bi-level search framework for training the policy and value networks. Experiments on five large real-world target graphs show that N-BLS can significantly improve the subgraph matching performance.

1 Introduction

With the growing amount of graph data that naturally arises in many domains, solving graph-related tasks via machine learning has gained increasing attention. Many NP hard tasks, e.g. Traveling Salesman Optimization [37], Graph Edit Distance computation [35], Maximum Common Subgraph detection [1], have recently been tackled via learning-based methods. These works on the one hand rely on search to enumerate the large solution space, and on the other hand use reinforcement learning (RL) to learn a good search policy from training data, thus obviating the need for hand-crafted heuristics adopted by traditional solvers. Such learning-to-search paradigm [1] also allows the training the RL agent without any solved instances for supervision. However, how to design a neural network architecture under the RL-guided search framework remains unclear for the task of Subgraph Matching, which requires the detection of all occurrences of a small query graph in an orders-of-magnitude larger target graph. Subgraph Matching has wide applications in graph database search [19], knowledge graph query [12], biomedical analysis [40], social group finding [23], quantum circuit design [9], etc. As a concrete example, Subgraph Matching is used for protein complex search in a protein-protein interaction network to test whether the interactions within a protein complex in a species are also present in other species [3].

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

Due to its NP-hard nature, the state-of-the-art Subgraph Matching algorithms rely on backtracking search with various techniques proposed to reduce the large search space [30, 11, 34]. However, these techniques are mostly driven by heuristics, and as a result, we observe that such solvers often fail to find any solution on large target graphs under a reasonable time limit, although they tend to work well on small graph pairs. We denote this phenomenon as *solution sparsity*. Such solution sparsity requires the designed model to not only have enough capacity but also to run efficiently under limited computational budget. Another consequence of solution sparsity is that, there can be little-to-no reward signals for the RL agent under a self-play training framework [29], which we denote as *reward sparsity*.

In this paper, we propose N-BLS (*Neural Bi-level Search*) with two means to address the aforementioned challenges. First, we propose a novel graph encoder-decoder neural network to dynamically match the query graph with the target graph and perform aggregation operation only on the query graph to reduce information loss. The novel encoder decouples the intra-graph message passing module (the "propagation" module) that yields state-independent node embeddings, and the inter-graph message passing module (the "matching" module) that refines the node embeddings via subgraph-to-graph matching. Thus, the intra-graph embeddings can be computed only once at the beginning of search for efficient inference. We further advance the inter-graph message passing by propagating only between nodes that either are already matched or can be matched in future by running a local candidate search space computation algorithm at each search state. Such algorithm leverages the key requirement of Subgraph Matching that every node and edge in the query graph must be matched to the target graph, and therefore reduces the amount of candidates from all the nodes in the target graph to a much smaller amount. Compared with a Graph Matching Network [20] which computes all the pairwise node-to-node message passing between two input graphs, our matching module is able to focus on only the node pairs that can contribute to the solution, and thus is both more effective and more efficient. Second, to address the reward sparsity issue incurred by the size of the target graph, we augment the backbone backtracking search algorithm with an inner Monte Carlo Tree Search (MCTS) to refine the raw policy predicted by the neural network, yielding an improved search policy. This results in a bi-level search algorithm, where the outer backtracking search and the inner MCTS work together for the policy and value training. Experiments on real graph datasets demonstrate that N-BLS outperforms baseline solvers in terms of effectiveness by a large margin. Our contributions can be summarized as follows:

- We address the challenging yet important task of Subgraph Matching with a vast amount of practical applications and propose N-BLS as the solution.
- One key novelty is a proposed encoder layer consisting of a propagation module and a matching module that dynamically passes the information between the input graphs.
- The adoption of MCTS results in a bi-level search framework for effective training the policy and value networks, which is another novelty.
- We conduct extensive experiments on real-world graphs to demonstrate the effectiveness of the proposed approach compared against a series of strong baselines in Subgraph Matching.

2 Preliminaries

2.1 Problem Definition

We denote a query graph as $q = (V_q, E_q)$ and a target graph as $G = (V_G, E_G)$ where V and E denote the node and edge sets. q and G are associated with a node labeling function L_g which maps every node into a label l in a label set Σ. **Subgraph** For a subset of nodes S of V_q, $q[S]$ denotes the subgraph of q with an node set S and an edge set consisting of all the edges in E_q that have both endpoints in S. **Subgraph isomorphism** q is subgraph isomorphic to G if there exists an injective node-to-node mapping $M : V_q \to V_G$ such that (1) $\forall u \in V_q, L_g(u) = L_g(M(u))$; and (2) $\forall e_{(u,u')} \in E_q, e_{(M(u),M(u'))} \in E_G$. **Subgraph Matching** The task of Subgraph Matching aims to find the subgraphs in G that are isomorphic to q. We call M a solution, or a match of q to G. We call a pair (q, G) is solved if the algorithm can find any match under a given time limit, which we find a challenge for existing solvers on input graphs in experiments.

2.2 Related Work

Non-learning methods on Subgraph Matching Existing methods on Subgraph Matching can be broadly categorized into backtracking search algorithms [27, 8, 7, 6, 11, 34] and multi-way join approaches [16, 17, 18, 10]. The former category of approaches employ a branch and bound approach to grow the solution from an empty subgraph by gradually seeking one matching node pair at a time following a strategic order until the entire search space is explored. The multi-way join approaches rely on decomposing the query graph into nodes and edges and performing join operations repeatedly to combine the partially matched subgraphs to q. However, they tend to work well on small query graphs generally with less than 10 nodes [30], and thus we follow and compare against methods in the former category, whose details will be shown in Section 2.3.

Learning-based methods for subgraph-related problems The idea of designing neural networks for predicting subgraph-graph relation has been explored, which due to its approximate nature may contain false positives/negatives, and it remains unclear how one can directly use these methods for exact Subgraph Matching. For example, NMATCH [22] learns node embeddings to predict a score for an input subgraph-graph pair indicating whether the subgraph is contained in another graph, which can be regarded as solving the approximate version of Subgraph Matching. Another direction of research aims to perform subgraph counting [21, 5] supervised on the number of specific substructures, and again lacks an explicitly search strategy and thus falls short of yielding solutions for Subgraph Matching.

Efforts on using RL for graph NP-hard problems The idea of using RL to replace heuristics in search algorithms for NP-hard graph-related tasks is not new, and we identify three works similar to the present work. (1) GLSEARCH [1] detects the maximum common subgraph (MCS) in an input graph pair, which is different from Subgraph Matching which requires the entire q to be matched with G, allowing further improvement in the neural network and search design. (2) RL-QVO [34] tackles Subgraph Matching via ordering the nodes in the *query* graph as a global pre-processing step before search, which is an orthogonal direction to our approach to select nodes in the target graph computed at each search step. (3) BIHYB [35] is a general framework with an upper-level RL agent to modify the graph and a lower-level heuristic algorithm for tasks such as Graph Edit Distance computation. However, the method is not tailored addressing the special challenges of Subgraph Matching.

2.3 Search-based methods for Subgraph Matching

Due to the NP-hard nature of Subgraph Matching, backtracking search is a naturally suitable algorithm since it exhaustively explores the solution space by starting with an empty match and adding one new node pair to the current match at each step. When the current match cannot be further extended, the search backtracks to its previous search state, and explores other node pairs to extend the match. However, naively enumerating all the possible states in the entire search space is intractable in practice, and therefore existing efforts mainly aim to reduce the total number of search steps for the backtracking search via mainly three ways [30]: (1) Filter nodes in G to obtain a small set of candidate nodes for each node in q as a pre-processing step before the backtracking search; (2) Order the nodes in q before the search; (3) Generate a local candidate set of nodes in each step of the search based on the current search state. Algorithm 1 summarizes the overall backtracking search based framework for Subgraph Matching. It is worth noting that the first three means correspond to the three steps in the algorithm, and therefore any improvement in any of the three steps can be regarded as orthogonal to each other.

Algorithm 1 Search-based Subgraph Matching.

1: **Input:** Query graph q, data graph G.
2: **Output:** Matches from q to G.
3: Filter: $C \leftarrow$ generate candidate node sets.
4: Order: $\phi \leftarrow$ generate an ordering for V_q.
5: Search: `Backtracking`$(q, G, C, \phi, \{\})$.

The basic idea of backtracking search is outlined in Algorithm 2. The recursive algorithm starts with an empty node-node mapping, and tries to add one new node pair to the mapping M at each recursive call. The action is the new node pair (u_t, v_t), where $u_t \in V_q$ is selected according to the ordering ϕ, and $v_t \in V_G$ is selected according to a policy to be one of the local candidate nodes (line 8), that can be mapped to u_t. It is noteworthy that this local candidate node set "$\mathcal{A}_{u_t} \subseteq V_G$" is refined over the global candidate sets C based on the current search state s_t, defined as (q, G) along with the current

mapping $M.$ and u_t whose details are non-trivial and explained in the supplementary material, since it ensures any node in $\mathcal{A}_{u_t}$ would lead to the extended subgraphs at s_t still being isomorphic to each other. Thus, we regard the local candidate set $\mathcal{A}_{u_t}$ for u_t as our action space, for which we learn a policy (line 9) to order the nodes, resulting in an ordered list $\mathcal{A}_{u_t,\text{ordered}}$.

Despite existing efforts to compute a small C, a good ϕ, and a small $\mathcal{A}_{u_t}$ to reduce search space, we observe that the size of $\mathcal{A}_{u_t}$ can be up to thousands of nodes for many real-world large target graphs, calling for a smarter policy to order the nodes not only in q but also in G ($\mathcal{A}_{u_t}$ to be specific). To the best of our knowledge, all the existing methods adopt a random ordering, i.e. a $\mathcal{A}_{u_t,\text{ordered}}$, to select a node from $\mathcal{A}_{u_t}$. We conjecture it is because existing local candidate computation techniques can further prune nodes from C, and thus enumerating $\mathcal{A}_{u_t}$ in a random ordering can be tractable on small graph pairs. We will experimentally show that this accounts for their failure to find a match for many large graph pairs. In fact, a theoretically perfect policy, $policy^*$, can find the entire match of the query graph in V_q steps or recursive calls, assuming q has at least one match with G. This again motivate our proposed method to improve the policy over $\mathcal{A}_{u_t}$.

Algorithm 2 `Backtracking`(q, G, C, ϕ, M)

1: **Input:** q, G, C, ϕ, and current mapping M.
2: **Output:** Subgraph match mappings.
3: **if** $|M| = |V_q|$ **then**
4: output M;
5: return;
6: **end if**
7: $u_t \in V_q \leftarrow \phi(M)$;
8: $\mathcal{A}_{u_t} \leftarrow s_t.getLocalCand(u_t)$;
9: $\mathcal{A}_{u_t,\text{ordered}} \leftarrow policy(s_t, \mathcal{A}_{u_t})$;
10: **for** v_t in $\mathcal{A}_{u_t,\text{ordered}}$ **do**
11: $M \leftarrow M.add(u_t, v_t)$;
12: `Backtracking`(q, G, C, ϕ, M);
13: $M \leftarrow M.remove(u_t, v_t)$;
14: **end for**

3 Proposed Method

In this section we formulate the problem of Subgraph Matching as learning an RL agent that grows the extracted subgraphs by adding new node pairs to the current subgraphs. N-BLS runs an inner MCTS simulation under each state of the outer backtracking search, resulting in a bi-level search framework. We first describe the environment setup, then depict our proposed encoder-decoder neural network which provides actions for our agent to grow the subgraphs in a search context. Finally, we describe how to tackle the sparsity of solutions in Subgraph Matching via bi-level tree search.

3.1 RL Formulation of Subgraph Matching

We formulate the Subgraph Matching problem as a Markov Decision Process (MDP), where the RL agent starts with an empty solution, and iteratively matches one node pair at a time until no more nodes can be matched. As mentioned in Section 2.3, our policy assigns a score to each action in $\mathcal{A}_{u_t}$, and therefore can be modeled as a policy network $P_\theta(a_t|s_t)$ that computes a probability distribution over $\mathcal{A}_{u_t}$ for the current state s_t, where the node pair to select consists of (u_t, v_t). However, since u_t is chosen by ϕ, we regard an action as a node v_t selected from $\mathcal{A}_{u_t}$. Since each action is a target node in $\mathcal{A}_{u_t}$, the neural network must learn good node embeddings that capture the current matching status in each search state, which will be shown in Section 3.2. We will also show that a value network $v_\theta(s_t)$ is needed to facilitate the training of $P_\theta(a_t|s_t)$ in Section 3.3. We describe the definition of reward in the rest of this section.

Since the entire q is desired to be matched with G, a key requirement of Subgraph Matching, we define the immediate reward of matching one new node pair as a function of the current subgraph size: $r = \exp\big(\alpha\frac{|S|+1}{|q|}\big) - \exp(\alpha\frac{|S|}{|q|})$, where $|S|$ denotes the number of nodes in $q[S]$ (or $G[S]$), $|q|$ denotes the number of nodes in q, and α is a hyperparameter controlling the slope of the reward curve. This function starts by assigning a relatively small value initially when the partial solution is small, and encourages the RL agent to match more and more nodes towards larger solutions due to the exponential function. For simplicity, we define the accumulated reward function $\rho(s_t) = \exp(\alpha\frac{|S(s_t)|}{|q|})$, where $|S(s_t)| = |S_{t,q}| = |S_{t,G}|$ denotes the size of the matched subgraph at s_t. Under policy π, the value function is defined as $v_\pi(s_t) = \mathbb{E}_{\tau\sim\pi}[\sum_{j=0}^{T}\gamma^j r_{t+j+1}|s = s_t]$, where γ is the discount factor, and

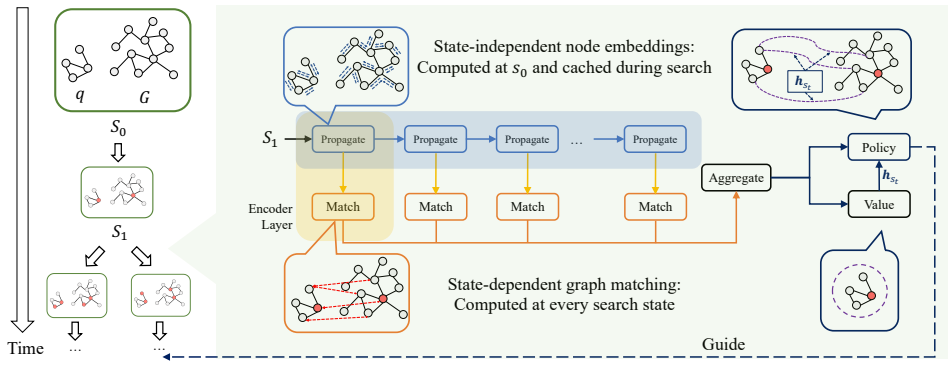


Figure 1: The overall process of Subgraph Matching is a bi-level search algorithm that matches one node pair at a time for the input query q and target graph G guided by a learned policy. Due to the large action space incurred by the large G in practice, we propose to train a policy network to guide the selection of local candidate nodes in G at each search state. This requires effective node embeddings to be learned that can reflect the node-to-node mapping at the current state and contribute to the prediction of the policy and reward. To achieve this end, we propose a novel QC-SGMNN encoder layer consisting of a "Propagation" module that performs intra-graph message passing as a typical GNN and a "Matching" module that performs state-dependent inter-graph matching (Section 3.2.1). A value network is also trained to facilitate the improvement of policy prediction under the Monte Carlo Tree Search self-play training framework (Section 3.3).

T denotes the total number of steps from s_t to a terminal state under π. In Subgraph Matching, the environment can be regarded as deterministic, i.e. s_{t+1} is determined based on s_t and a_t, and if we assume $\gamma = 1$, $v_\pi(s_t)$ can be simplified to $v_\pi(s_t) = \rho(s_T) - \rho(s_t)$, which intuitively reflects the eventual solution size starting from s_t.

3.2 Encoder-Decoder Design for Policy and Value Estimation

Our neural network consists of an encoder which produces node-level embeddings for V_q and V_G and a decoder which transforms the node embeddings into $P_\theta(a_t|s_t)$ and $v_\theta(s_t)$. We identify the following challenges: (1) Each state s_t consists of q, G, and the mapping M_t between the matched nodes $S_{t,q} \subseteq V_q$ and $S_{t,G} \subseteq V_G$, which should be effectively represented and utilized in predicting $v_\theta(s_t)$ and $P_\theta(a_t|s_t)$; (2) $P_\theta(a_t|s_t)$ is dependent on s_t and a_t, potentially requiring the node embeddings to be recomputed at every search step and incurring too much computational overhead; (3) Existing Graph Neural Networks (GNN) relying on message passing (e.g. [14, 33, 39]) are inherently local, which are incompatible with value and policy estimation requiring global matching of q to G. Even the more recent GNNs that enhance expressiveness beyond 1-Weisfeiler-Lehman test [25, 36] still cannot guarantee the capturing of nodes in the solution, since otherwise the NP-hard task would then be solved. This motivates the following design leveraging the properties of Subgraph Matching as much as possible.

3.2.1 Encoder: Embedding Generation via QC-SGMNN

The nodes of q and G are encoded into initial one-hot embeddings $\boldsymbol{h}^{(0)} \in \mathbb{R}^{|\Sigma|}$, and fed into K sequentially stacked Query-Conditioned Subgraph Matching Neural Networks (QC-SGMNN) layers to obtain $\boldsymbol{h}^{(K)} \in \mathbb{R}^D$, where D is the dimension of embeddings, followed by a JUMPING KNOWLEDGE [38] network to combine the node embeddings from the K layers to obtain the final node embeddings, denoted as "Aggregate" in Figure 1.

We start by observing that Subgraph Matching desires all nodes in q to be matched with G, and existing works in Subgraph Matching aiming at reduce $\mathcal{A}_{u_t}$ (Section 2.3) guarantee no false pruning. In other words, $\mathcal{A}_{u_t}$ is guaranteed to contain the solution nodes in G that should match with u_t, i.e. $\mathcal{A}_{u_t}$ provides useful hints of which nodes should be matched together. We then compute the candidate space $\mathcal{A}_{u'}$ for every remaining query graph node $u' \notin S_{t,q}$, resulting in a mapping $M'_t = \{u' \mapsto \mathcal{A}_{u'}, \forall u' \in V_q \setminus S_{t,q}\}$. Intuitively, M'_t can be regarded as hints of *future* node-node mappings. In contrast, $M_t = \{u \mapsto \{v\}, \forall u \in S_{t,q}\}$ reflects the *current* state s_t, which maps each

node in $S_{t,q}$ to a unique node in G. We define $\tilde{M}_t$ as the union of M_t and M'_t, and define $\tilde{M_t}^{-1}$ as the reverse mapping of $\tilde{M}_t$, which serve as the basis for the matching module.

QC-SGMNN consists of a propagation module, which performs regular intra-graph message passing on q and G individually, and a matching module, which performs state-dependent subgraph-graph matching leveraging $\tilde{M}_t$ to pass information between q and G. Specifically,

$$\begin{aligned}
\boldsymbol{h}_{u,\text{intra}}^{(k+1)} &= f_{\text{agg}}\big(\{f_{\text{msg}}(\boldsymbol{h}_{u,\text{intra}}^{(k)}, \boldsymbol{h}_{u',\text{intra}}^{(k)}) | u' \in \mathcal{N}(u)\}\big), \\
\boldsymbol{h}_{v,\text{intra}}^{(k+1)} &= f_{\text{agg}}\big(\{f_{\text{msg}}(\boldsymbol{h}_{v,\text{intra}}^{(k)}, \boldsymbol{h}_{v',\text{intra}}^{(k)}) | v' \in \mathcal{N}(v)\}\big), \\
\boldsymbol{h}_{q,\text{intra}}^{(k+1)} &= f_{\text{readout}}(\{\boldsymbol{h}_{u,\text{intra}}^{(k+1)} | u \in V_q\}), \\
\boldsymbol{h}_{u,G\to q}^{(k+1)} &= f_{\text{agg}}\big(\{f_{\text{msg}}(\boldsymbol{h}_{u,\text{intra}}^{(k+1)}, \boldsymbol{h}_{v,\text{intra}}^{(k+1)}) | v \in \tilde{M}_t(u)\}\big), \\
\boldsymbol{h}_{v,q\to G}^{(k+1)} &= f_{\text{agg}}\big(\{f_{\text{msg}}(\boldsymbol{h}_{v,\text{intra}}^{(k+1)}, \boldsymbol{h}_{u,\text{intra}}^{(k+1)}, \boldsymbol{h}_{q,\text{intra}}^{(k+1)}) | u \in \tilde{M_t}^{-1}(v)\}\big), \\
\boldsymbol{h}_{u}^{(k+1)} &= f_{\text{combine}}(\boldsymbol{h}_{u,G\to q}^{(k+1)}, \boldsymbol{h}_{u,\text{intra}}^{(k+1)}), \\
\boldsymbol{h}_{v}^{(k+1)} &= f_{\text{combine}}(\boldsymbol{h}_{v,q\to G}^{(k+1)}, \boldsymbol{h}_{v,\text{intra}}^{(k+1)}).
\end{aligned} \tag{1}$$

The first two steps can be any intra-graph message passing GNNs such as Graph Attention Networks [33] with a message function f_{msg} and an aggregation function f_{agg}, corresponding to the propagation module. The middle three steps compute intermediate embeddings that will be used for the last two steps, i.e. the matching module. Specifically, $\boldsymbol{h}_{u,G\to q}^{(k+1)}$ and $\boldsymbol{h}_{v,q\to G}^{(k+1)}$ compute the cross-graph message passing from G to q and q to G using $\tilde{M}_t$ and $\tilde{M_t}^{-1}$, respectively (represented as the red dashed lines in Figure 1). A graph-level embedding $\boldsymbol{h}_{q,\text{intra}}^{(k)}$ is computed via f_{readout} and used in the information passing from q to G to let the embeddings of V_G query-conditioned. We do not inject the graph-level embeddings of G into $\boldsymbol{h}_{u,G\to q}^{(k+1)}$, since the large size of G could result in too much information loss in the readout operation. The last two steps combine the intra-graph embeddings and inter-graph embeddings via f_{combine} to produce the final output embeddings. It is noteworthy that f_{msg} and f_{agg} refer to the general class of functions that yield messages between two nodes and performs aggregation on a set of messages, and in practice, the propagation and matching modules can use different functions for f_{msg} and f_{agg}.

To address challenge (2), we make the observation that $\boldsymbol{h}_{\text{intra}}^{(k)}$ only depends on q and G and is independent of M_t, and thus can be computed once and cached at the beginning of the search (denoted as the top branch in Figure 1) and later reused throughout the search. Our QC-SGMNN decouples the propagation and matching steps and outputs two sources of information separately, allowing the search to cache the state-independent node embeddings and dynamically select the computational paths during search. Thanks to the caching, only the initial iteration requires the $\mathcal{O}(|E_q|+|E_G|)$ computation, and all the subsequent iterations only involve $\mathcal{O}\big(|V_q||\bar{\mathcal{A}}_{u_t}|\big)$ complexity, where $|\bar{\mathcal{A}}_{u_t}|$ is the average size of local candidate space.

3.2.2 Decoder: $v_\theta(s_t)$ and $P_\theta(a_t|s_t)$ Estimation

Since the node embeddings of q has received the right amount of information from $\tilde{M}_t$, we propose an attention-based mechanism to compute the value: $v_\theta(s_t) = \text{MLP}(\boldsymbol{h}_{s_t}) = \text{MLP}\big(\sum_{u\in V_q} f_{\text{att}}(\boldsymbol{h}_u, \{\boldsymbol{h}'_u | u' \in V_q\})\boldsymbol{h}_u\big)$, where f_{att} computes one attention score per node normalized across V_q to tackle challenge (3). Intuitively, the attention function learns which nodes are important for contributing to the eventual subgraph selected by s_T. Due to the cross-graph communication in QC-SGMNN, we only aggregate nodes from V_q to obtain the state representation $\boldsymbol{h}_{s_t}$, taking advantage of the fact that $|V_q|$ is typically much smaller than $|V_G|$ in Subgraph Matching, further addressing challenge (2).

For the policy, we again aim to tackle challenge (3), by reusing $\boldsymbol{h}_{s_t}$. The reasons are two-fold. First, by definition $P_\theta(a_t|s_t)$ requires s_t as input; Second, the attention mechanism used to compute $\boldsymbol{h}_{s_t}$ capturing the future subgraph. Combined with a bilinear tensor product with learnable parameter $\boldsymbol{W}^{[1:F]} \in \mathbb{R}^{D\times D\times F}$ with a hyperparameter F to allow the action node embeddings $\boldsymbol{h}_{u_t}$ and $\boldsymbol{h}_{v_t}$ to fully interact, we obtain $P_{\text{logit}}(a_t|s_t) = \text{MLP}\big(\text{CONCAT}(\boldsymbol{h}_{u_t}^T \boldsymbol{W}^{[1:F]} \boldsymbol{h}_{v_t}, \boldsymbol{h}_{s_t})\big)$, followed by a softmax normalization over the logits for all the actions. The decoder has time complexity $\mathcal{O}\big(|V_q| + |\bar{\mathcal{A}}_{u_t}|\big)$.

3.3 Addressing Reward Sparsity via Monte Carlo Tree Search (MCTS)

To address the reward sparsity issue mentioned in Section 1, we adopt the Monte Carlo Tree Search (MCTS) method [4, 15] used in AlphaZero [28] and MuZero [26] as an additional step to improve the raw policy predicted by a neural network. We conjecture MCTS helps alleviating the reward sparsity issue due to its ability to spend additional computation at each s_t to further explore and revise $P_\theta(a_t|s_t)$, and provide a better search policy π based on the visiting counts associated with each state. We thus let N-BLS perform MCTS at each search step during training. Since MCTS naturally expands s_t into a search tree below it, by performing MCTS at each s_t, we obtain a bi-level search framework where the outer level is the backtracking search, which always selects a new node pair to match, and the inner level is the MCTS, which allows revisiting states and providing an improved search policy π used for training $P_\theta(a_t|s_t)$. We only use the outer search during inference to make it as fast as possible.

We find for our task, the varying number of actions per state causes insufficient exploration for states with a large action space, while too much exploration for states with a small action space. We propose to dynamically compute how many MCTS iterations to perform for each state based on the action space size, $|\mathcal{A}_{u_t}|$, using $\min(\omega|\mathcal{A}_{u_t}|, \max_\text{iters})$. In other words, the amount of MCTS iterations allocated for each state during training is proportional to the action space size bounded by a pre-defined maximum number of iterations allowed, $\max_\text{iters}$. More details can be found in the supplementary material.

In order to utilize MCTS, a value network $v_\theta(s_t)$ needs to be designed to facilitate the selection step of each MCTS iteration. Initially, it is unlikely that the untrained $v_\theta(s_t)$ and $P_\theta(a_t|s_t)$ lead to a large reward $v_\pi(s_t)$ by following π, but since the reward is used to train $v_\theta(s_t)$, in subsequent training epochs, the improved $v_\theta(s_t)$ can guide MCTS to find better π, and thus a better $P_\theta(a_t|s_t)$. Overall, both the policy and value networks are trained under the self-play framework. More explanations and intuitions can be found in the supplementary material. We use the loss function in [29] to train $P_\theta(a_t|s_t)$ and $v_\theta(s_t)$: $L = \big(v_\pi(s_t) - v_\theta(s_t)\big)^2 - \boldsymbol{\pi}^T \log(\boldsymbol{P}) + c||\boldsymbol{\theta}||^2$, where c is a hyperparameter for weight regularization and $\boldsymbol{\theta}$ denotes the parameters of our model.

4 Experiments

We evaluate N-BLS against five backtracking-based algorithms for exact Subgraph Matching, and conduct experiments on five real-word target graphs from various domains, whose details can be found in the supplementary material. We find N-BLS written in Python can outperform the state-of-the-art solver written in C++ under most cases, suggesting the effectiveness of the learned policy for ordering nodes in G. Code, trained model, and all the datasets used in the experiments are released as part of the supplementary material for reproducibility. More results and the ablation study can be found in the supplementary material.

4.1 Datasets and Evaluation Protocol

We use five large real-world target graphs provided by a recent survey paper that consistently compares several Subgraph Matching solvers [30]. For each G, we generate three query sets for testing, each with 100 randomly sampled query graphs of 32, 64, and 128 nodes, denoted as "Q_{32}", "Q_{64}", and "Q_{128}", respectively, resulting in 100 (q, G) graph pairs. As shown in Table 1, the largest target graph, YOUTUBE, has over 1M nodes and 2M edges.

Table 1: Datasets description.

Dataset	Domain	$\|V_G\|$	$\|E_G\|$
HUMAN	Biology	4,674	86,282
HPRD	Biology	9,460	34,998
DBLP	Social	317,080	1,049,866
EU2005	Web	862,664	16,138,468
YOUTUBE	Social	1,134,890	2,987,624

To train our model, due to the self-play framework, we can sample query graphs of any sizes and simply let N-BLS perform the proposed bi-level search algorithm and collect training data from the search tree. Therefore, we sample query graphs from each target graph of sizes ranging from 4 nodes to 128 nodes, and validate that no query graph in the testing set is visible to the model, that is, no graph in the training set is isomorphic to any graphs in the testing set. During the testing stage, for each target graph, we use the following evaluation protocol. For each graph pair, we set a time limit

Table 2: Results on Subgraph Matching over five datasets each with three query sets. The ratio of the number of solved pairs over the 100 pairs per target graph is reported.

Dataset	QUICKSI	GRAPHQL	CECI	DP-ISO	HYBRID	NMATCH	N-BLS
HUMAN Q_{32}	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
HUMAN Q_{64}	0.07	0.19	0.07	0.06	0.10	**1.00**	0.99
HUMAN Q_{128}	0.02	0.02	0.00	0.00	0.02	0.32	**0.43**
HPRD Q_{32}	0.52	0.53	0.65	0.55	0.70	0.90	**0.99**
HPRD Q_{64}	0.02	0.06	0.06	0.25	0.06	0.06	**0.59**
HPRD Q_{128}	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DBLP Q_{32}	0.74	0.43	0.89	0.64	0.70	**1.00**	**1.00**
DBLP Q_{64}	0.40	0.14	0.59	0.44	0.29	0.71	**0.97**
DBLP Q_{128}	0.07	0.02	0.18	0.16	0.04	0.04	**0.58**
EU2005 Q_{32}	0.24	0.06	0.22	0.26	0.18	0.82	**1.00**
EU2005 Q_{64}	0.02	0.00	0.02	0.03	0.01	0.26	**0.63**
EU2005 Q_{128}	0.01	0.01	0.01	0.01	0.01	0.01	**0.21**
YOUTUBE Q_{32}	0.35	0.24	0.40	0.38	0.53	0.41	**0.78**
YOUTUBE Q_{64}	0.26	0.23	0.37	0.36	0.24	0.10	**0.90**
YOUTUBE Q_{128}	0.04	0.02	0.04	0.10	0.02	0.00	**0.11**

of 5 minutes, and record the result of whether a solution is found or not, and then average across the 100 pairs for each target graph.

4.2 Baselines and Parameter Settings

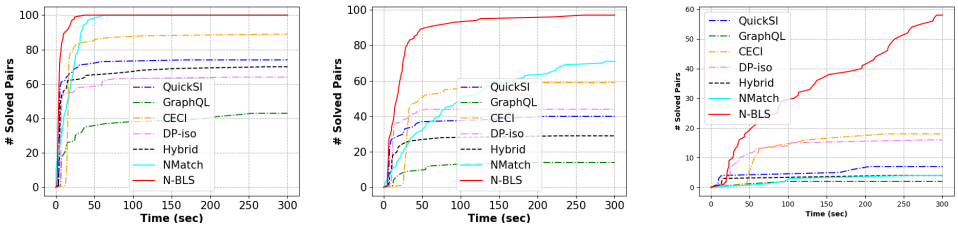
We compare N-BLS against a series of baseline solvers written in C++ whose source codes are provided by [30]: QUICKSI [27], GRAPHQL [8], CECI [2], DP-ISO [6], and HYBRID [30]. For HYBRID, we follow the recommendation by the authors of [30] by using DP-ISO for filtering, GRAPHQL for query node ordering, and LFTJ [32, 31] for local candidate computation. We also include NMATCH [22] as a learning-based method for exact Subgraph Matching, by adapting the original implementation provided by the authors of [22] into our backtracking search framework. Specifically, at each search step, we invoke the inference step of the released trained NMATCH model over the input graphs, and obtain a matching score for each (u_t, v_t) action node pair to guide the selection of nodes in $\mathcal{A}_{u_t}$.

Since we are among the first to learn a policy for selecting nodes in G, we use DP-ISO for filtering and GRAPHQL for query node ordering. We implement the backtracking search framework and DP-ISO's local candidate computation algorithm in Python. N-BLS uses 4 layers of the proposed QC-SGMNN encoders. ω is set to 10 and $\mathrm{max_iters}$ is set to 120. Training is performed on a server with NVIDIA Tesla V100 GPUs. We train N-BLS for 2 days with the Adam optimizer [13] and initial learning rate 0.001.

4.3 Results

As shown in Table 2, N-BLS achieves the same or better performance on 14 out of 15 query sets, across a variety of query graph sizes, target graph sizes, graph densities, and domains. We observe baseline solvers fail completely on many Q_{128} and Q_{64} query sets, as search space pruning on its own is not effective enough to guarantee solutions in such cases. Learning-based methods outperform solver baselines because they provide a better target graph node ordering, under the same search framework. Out of the learning-based methods, N-BLS consistently outperforms NMATCH, as it features a more powerful encoder that efficiently computes a policy conditioned on the current search state with a better inter-graph communication mechanism. This also suggests that the target graph node ordering provided by N-BLS is much better than the random ordering adopted by solver baselines, confirming that our novelties indeed greatly improve Subgraph Matching performance.

We observe that when the query graph size increase, all methods show lower performance, which can be attributed to the exponentially growing search space. It is noteworthy that the survey paper comparing existing solvers [30] uses query graphs up to 32 nodes, whereas we challenge all methods by testing on query graphs up to 128 nodes. The fact that N-BLS is able to solve more graph pairs than baselines when the query graphs contain 128 nodes demonstrates good scalability of N-BLS.



(a) **DBLP** Q_{32} (b) **DBLP** Q_{64} (c) **DBLP** Q_{128}

Figure 2: Comparison of the curves of the number of solved pairs versus time on DBLP.

On HPRD Q_{32}, although NMATCH is able to solve 1 more pair than N-BLS, the fact that NMATCH is also a learning-based model and it outperforms the non-learning solvers suggests the effectiveness of using learning to select nodes in the target graph G. It is also worth noting that all methods fail to solve any pair on HPRD Q_{128}. A recent work [24] studying when Subgraph Matching is difficult empirically discovers that sometimes small query and target graphs may also be hard to solve. Such hard instances can be identified using a phase transition plot, where the difficulty of different graph pairs is visualized by running a Subgraph Matching solver on many randomly generated graph pairs with varying edge probabilities. However, how to estimate the difficulty of a given graph pair without running a solver remains an open question. Regardless of how to locate such difficult graph pairs, we still verify the usefulness of learning-based approaches. N-BLS can be beneficial not only on large target graphs, but also smaller target pairs that existing algorithms struggle to solve. We indeed see this trend on HUMAN Q_{128}, the smallest of the 5 target graphs, where baselines fail to solve few if any graph pairs and N-BLS solves many. This suggests our model can enable Subgraph Matching applications on previously unattainable datasets.

4.4 Visualization of Performance across Time

To examine the efficiency of each method and further analyze the efficacy across time, we conduct the following experiment. For each (q, G) pair, we record the time the method takes to find a solution, and accumulate the number of solved pairs across time. From $t = 0$ to $t = 300$ seconds, an increase at t indicates the method solves one graph pair at t. The earlier a method reaches 100 pairs, the faster and better the method is. As seen in Figure 2, N-BLS runs fast on Q_{32} and takes more time on Q_{64} and Q_{128} to solve a pair, but consistently outperforms baselines at the end of 5 minutes, demonstrating the efficiency of N-BLS in practice.

In theory, given an infinite amount of time, every method adopting the backtracking search algorithm would be able to solve a graph pair. However, such assumption is not practical. Thus, the curves in Figure 2 have the practical implication that under a reasonable amount of time budget, the idea of using a smarter policy indeed brings better performance to Subgraph Matching. Another observation is that the baseline solvers flatten towards the end of 5 minutes, indicating that they get stuck in unpromising search states that are unlikely to contain the solution, confirming the severity of the aforementioned challenges of solution and reward sparsity. It is worth noting that N-BLS on Q_{128} continues to solve more pairs towards the end of 5 minutes, suggesting that further improvement may be achieved if N-BLS is implemented in C++ with optimization.

5 Conclusion

In this paper, we tackle the challenging and important task of Subgraph Matching, and present a new method for efficient and effective exact Subgraph Matching. It contains a novel encoder-decoder neural network architecture trained under by adopting the Monte Carlo Tree Search (MCTS) algorithm for self-play to improve the policy and value estimation. The core component, QC-SGMNN, is a query-conditioned graph encoder that performs intra-graph propagation and inter-graph node matching to capture each state. To address the reward sparsity issue posed by the large action space, we perform MCTS at each search iteration to obtain more reliable training signals. We experimentally show the practical usefulness of the proposed N-BLS method on the important Subgraph Matching task. Specifically, N-BLS is able to solve more graph pairs than several existing Subgraph Matching solvers on five large real-world datasets.

References

[1] Yunsheng Bai, Derek Xu, Yizhou Sun, and Wei Wang. Glsearch: Maximum common subgraph detection via learning to search. In *ICML*, pages 588–598. PMLR, 2021.

[2] Bibek Bhattarai, Hang Liu, and H Howie Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1447–1462, 2019.

[3] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 14(7):1–13, 2013.

[4] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[5] Zhengdao Chen, Lei Chen, Soledad Villar, and Joan Bruna. Can graph neural networks count substructures? *NeurIPS*, 33:10383–10395, 2020.

[6] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1429–1446, 2019.

[7] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 337–348, 2013.

[8] Huahai He and Ambuj K Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418, 2008.

[9] Hui Jiang, Yuxin Deng, and Ming Xu. Quantum circuit transformation based on subgraph isomorphism and tabu search. *arXiv e-prints*, pages arXiv–2104, 2021.

[10] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1695–1698, 2017.

[11] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of the 2021 International Conference on Management of Data*, pages 925–937, 2021.

[12] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. Taming subgraph isomorphism for rdf query processing. *arXiv preprint arXiv:1506.01973*, 2015.

[13] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2015.

[14] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *ICLR*, 2016.

[15] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[16] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment*, 8(10):974–985, 2015.

[17] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 10(3):217–228, 2016.

[18] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment*, 12(10):1099–1112, 2019.

[19] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment*, 6(2):133–144, 2012.

[20] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. *ICML*, 2019.

[21] Xin Liu, Haojie Pan, Mutian He, Yangqiu Song, Xin Jiang, and Lifeng Shang. Neural subgraph isomorphism counting. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1959–1969, 2020.

[22] Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, Jure Leskovec, et al. Neural subgraph matching. *arXiv preprint arXiv:2007.03092*, 2020.

[23] Tinghuai Ma, Siyang Yu, Jie Cao, Yuan Tian, Abdullah Al-Dhelaan, and Mznah Al-Rodhaan. A comparative study of subgraph matching isomorphic methods in social networks. *IEEE Access*, 6:66621–66631, 2018.

[24] Ciaran McCreesh, Patrick Prosser, Christine Solnon, and James Trimble. When subgraph isomorphism is really hard, and why this matters for graph databases. *Journal of Artificial Intelligence Research*, 61:723–759, 2018.

[25] Christopher Morris, Gaurav Rattan, and Petra Mutzel. Weisfeiler and leman go sparse: Towards scalable higher-order graph embeddings. *NeurIPS*, 33:21824–21840, 2020.

[26] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[27] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 2008.

[28] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[29] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[30] Shixuan Sun and Qiong Luo. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1083–1098, 2020.

[31] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. Rapidmatch: a holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment*, 14(2):176–188, 2020.

[32] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012.

[33] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *ICLR*, 2018.

[34] Hanchen Wang, Ying Zhang, Lu Qin, Wei Wang, Wenjie Zhang, and Xuemin Lin. Reinforcement learning based query vertex ordering model for subgraph matching. *arXiv preprint arXiv:2201.11251*, 2022.

[35] Runzhong Wang, Zhigang Hua, Gan Liu, Jiayi Zhang, Junchi Yan, Feng Qi, Shuang Yang, Jun Zhou, and Xiaokang Yang. A bi-level framework for learning to solve combinatorial optimization on graphs. *NeurIPS*, 34, 2021.

[36] Asiri Wijesinghe and Qing Wang. A new perspective on" how graph neural networks go beyond weisfeiler-lehman?". In *ICLR*, 2022.

[37] Zhihao Xing and Shikui Tu. A graph neural network assisted monte carlo tree search approach to traveling salesman problem. *IEEE Access*, 8:108418–108428, 2020.

[38] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. *ICML*, 2018.

[39] Jiaxuan You, Zhitao Ying, and Jure Leskovec. Design space for graph neural networks. *NeurIPS*, 33:17009–17021, 2020.

[40] Shijie Zhang, Shirong Li, and Jiong Yang. Gaddi: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 192–203, 2009.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] See the supplementary material.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]