Learning to Accelerate Heuristic Searching for Large-Scale Maximum Weighted b-Matching Problems in Online Advertising

Xiaotian Hao[1] , **Junqi Jin**[2 *] , **Jianye Hao**[1,3,4 *] , **Jin Li**[2] , **Weixun Wang**[1] , **Yi Ma**[1] , **Zhenzhe Zheng**[5] , **Han Li**[2] , **Jian Xu**[2] and **Kun Gai**[2]
[1]College of Intelligence and Computing, Tianjin University
[2]Alibaba Group, Beijing
[3]Noah's Ark Lab, Huawei
[4]Tianjin Key Lab of Machine Learning
[5]Shanghai Jiao Tong University
{xiaotianhao,jianye.hao,wxwang, mayi}@tju.edu.cn, zhengzhenzhe220@gmail.com,
{junqi.jjq,echo.lj,lihan.lh,xiyu.xj}@alibaba-inc.com, jingshi.gk@taobao.com

Abstract

Bipartite b-matching is fundamental in algorithm design, and has been widely applied into economic markets, labor markets, etc. These practical problems usually exhibit two distinct features: large-scale and dynamic, which requires the matching algorithm to be repeatedly executed at regular intervals. However, existing exact and approximate algorithms usually fail in such settings due to either requiring intolerable running time or too much computation resource. To address this issue, we propose `NeuSearcher` which leverages the knowledge learned from previously instances to solve new problem instances. Specifically, we design a multichannel graph neural network to predict the threshold of the matched edges weights, by which the search region could be significantly reduced. We further propose a parallel heuristic search algorithm to iteratively improve the solution quality until convergence. Experiments on both open and industrial datasets demonstrate that `NeuSearcher` can speed up 2 to 3 times while achieving exactly the same matching solution compared with the state-of-the-art approximation approaches.

1 Introduction

Bipartite b-matching is one of the fundamental problems in computer science and operations research. Typical applications include resource allocation problems, such as job/server allocation in cloud computing and product recommendation[De Francisci Morales *et al.*, 2011] and advertisement (ad) allocation [Agrawal *et al.*, 2018] in economic markets. It has also been utilized as an algorithmic tool in a variety of domains, including document clustering [Dhillon, 2001], computer vision [Zanfir and Sminchisescu, 2018], and as a subroutine in machine learning algorithms. The focus of this paper is on large-scale real-world bipartite b-matching problems, which usually involve billions of nodes and edges and the graph structure dynamically evolves. One concrete example is the ads allocation in targeted advertising.

In targeted advertising, a bipartite graph connects a large set of consumers and a large set of ads. We associate a relevance score (e.g., click through rate) to each potential edge of a consumer to an ad, which measures the degree of interest a consumer has over an ad. Each edge then can be seen as an allocation from an ad to a consumer with the corresponding score. Due to the business reasons, for each consumer and ad, there are cardinality constraints on the maximum number of edges that each vertex can be allocated. The goal of the ad allocation is to search for a maximum weighted b-matching: selecting a subset of edges with the maximum total scores while satisfying the cardinality constraints.

The first exact algorithm for b-matching was the Blossom algorithm [Edmonds, 1965]. After that, several exact b-matching approaches have been proposed, such as branch and cut approach [Padberg and Rao, 1982], cutting plane technique [Grötschel and Holland, 1985] and belief propagation [Bayati *et al.*, 2011]. Interested readers can refer to [Müller-Hannemann and Schwartz, 2000] for a complete survey. The time complexity of these exact matching algorithms is proportional to the product of the numbers of edges and vertices [Naim and Manne, 2018]. In advertising, there exist hundreds of millions of consumers and ads with billions of edges, which makes the exact algorithms computationally infeasible.

Another challenge in advertising is that the bipartite graph dynamically evolves with time, e.g., consumers' interests over ads may be different in different period, which changes the edges' scores. For this reason, the matching problem has to be repeatedly solved (e.g., hour-to-hour) to guarantee matching performance. This requires that an algorithm must compute the solution fast to satisfy the online requirements. Though we can use approximate algorithms with parallel computation to reduce the new solution computation time [De Francisci Morales *et al.*, 2011; Khan *et al.*, 2016], all of them starts the solution computation of each new problem instance from scratch. It would be more desirable if the knowledge learned from previous solved instances can be (partially) transferred to the new ones (similar but not exactly the same) to further reduce the computation time.

For this purpose, we investigate whether we can leverage the representation capability of neural networks to transfer the knowledge learned from previous solved instances to ac-

*Corresponding authors.

celerate the solution computing on similar new instances. In this paper, we propose a parallelizable and scalable learning based framework `NeuSearcher` to accelerate the solution computing for large-scale b-matching. Our contributions in this paper can be summarized as follows: (1) We propose `NeuSearcher` which integrates machine learning to transfer knowledge from previous solved instances to similar new ones, which significantly reduces the computational cost and reaches up to 2-3 times faster than the state-of-the-art approximation algorithms. (2) We build a predictive model to predict the threshold of matched edges weights to reduce the search region of the solution space. Then, we design a heuristic search algorithm to ensure the solution quality and convergence. We show that it is guaranteed that the `NeuSearcher`'s solution quality is exactly the same with the state-of-the-art approximation algorithms. (3) As the bipartite graph in advertising is unbalanced, i.e., the number of consumers is extremely larger than that of ads, we design a multichannel graph neural network (GNN) to improve the accuracy of the predictive model. (4) Experiments on open and industrial large-scale datasets demonstrate that `NeuSearcher` can compute nearly optimal solution much faster than the state-of-the-art approaches.

2 Maximum Weighted b-Matching Problem

In a targeted advertising system, there are a set of ads $\mathbb{A}$=$\{a_1, ..., a_m\}$, which are to be delivered to a set of consumers $\mathbb{C}$=$\{c_1, ..., c_n\}$. For each a_i and c_j, we measure the interest of consumer c_j in ad a_i with a positive weight $\mathrm{w}(a_i, c_j)$ (e.g., click through rate). Each ad has to pay a fee to the platform once been displayed to (or clicked by) a consumer. Since the advertising budget is limited, each advertiser aims to pick out a limited number of their best audiences from $\mathbb{C}$ to deliver its ad to maximize the profits. Hence, we set a capacity constraint $b(a_i)$ on the number of consumers that each ad a_i can match. Besides, to avoid each consumer c_j receiving too many ads, we enforce a capacity constraint $b(c_j)$ on the number of ads that are matched to c_j. The goal is to find a subset of edges $M \subseteq \mathbb{E}$ such that the capacity constraints for each ad and consumer are satisfied, while maximizing the total weight of the matching. Such an edge set M is referred to as a maximum weighted b-matching. Formally, we have:

$$\max_{\mathcal{X}} \sum_{(a_i,c_j)\in\mathbb{E}} x_{i,j}\mathrm{w}(a_i, c_j) \tag{1}$$

$$\text{s.t.} \sum_{c_j\in\mathbb{C}} x_{i,j} \leq b(a_i), \forall a_i \in \mathbb{A}, \tag{2}$$

$$\sum_{a_i\in\mathbb{A}} x_{i,j} \leq b(c_j), \forall c_j \in \mathbb{C} \tag{3}$$

where $\mathcal{X}=\{x_{i,j}|(a_i, c_j) \in \mathbb{E}\}$ is the decision variable, $x_{i,j} \in \{0, 1\}$ indicates whether edge (a_i, c_j) is included in M.

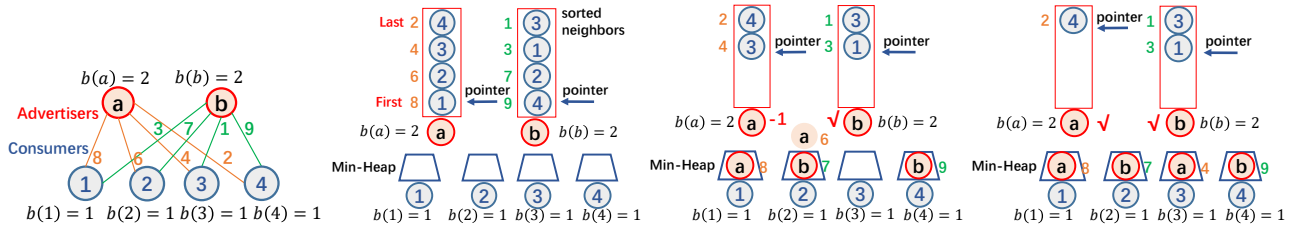
However, the relationship between consumers and advertisers changes frequently in practice. The main reason is that the consumers' interests are evolving, which changes the edge weight $\mathrm{w}(a_i, c_j)$ of the matching problem. Therefore, similar problem instances have to be repeatedly solved for better matching qualities. In the following of this paper, we consider these repeatedly solved b-matching problem instances $\mathcal{I} = \{I_1, ..., I_N\}$ are generated from the same distribution $\mathbb{D}$. And we are interested in investigating whether we can leverage neural network to transfer the knowledge learned from previous solved instances to accelerate the solution computing on new instances. Though, some recent works incorporate machine learning methods to solve combinatorial optimization (CO) problems, e.g., learning to solve the Traveling Salesman Problem [Vinyals *et al.*, 2015; Khalil *et al.*, 2017; Li *et al.*, 2018] and Mixed Integer Programming problems [He *et al.*, 2014; Chen and Tian, 2019; Ding *et al.*, 2019], no researches aim to solve the practical large-scale b-matching problems and these existing methods are not applicable in our case. The reason is that these methods usually model the CO problem as a sequential decision-making process via imitation learning or reinforcement learning, whose time complexity is proportional to the edge number. The time complexity is too high. Besides, these approaches can only be applied to small problem instances, e.g., problems with thousands nodes or edges. But the problem we consider is in billion scale.

Next, we start by analyzing the core idea and the bottlenecks of the state-of-the-art parallel approximation approaches. Then, we derive which form of knowledge can be transferred from previous solved problem instances to new ones and propose our `NeuSearcher` framework.

3 Bottleneck of Approximation Approaches

The greedy algorithm is the most commonly used approximation approach in practice. It works by sorting all the edges globally in descending order of their weights. After that, it picks edges one by one from the heaviest to the lightest only if the capacity constraints on both end points of an edge are satisfied. But, if the graph has billions of edges: (1) the global sorting of all edges costs too much time and becomes a bottleneck. (2) the sequential nature of adding edges to the solution is slow. Accordingly, paralleled greedy approaches are proposed, e.g., GreedyMR and b-suitor [Khan *et al.*, 2016; Naim and Manne, 2018], which are the state-of-the-art parallelizable approximate methods for computing b-matching solutions. We explain the core idea of these methods through a simple example. As shown in Fig 1(a), there are 2 ad vertices a and b, both of which have a capacity constraint $b(a)$ = $b(b)$ = 2. There are 4 consumer vertices whose indices range from 1 to 4, all of which have a constraint $b(1)$ =,...,= $b(4)$ = 1. And there is a weight $\mathrm{w}(a_i, c_j)$ marked alongside each edge (i.e., 3,7,1,9 in green and 8,6,4,2 in orange). The paralleled greedy approach works iteratively as:

- At the initial step (Figure 1(b)), each consumer c initializes an empty minimum heap of size $b(c)$=1 (shown as blue trapezoids). The target is to reserve the top-$b(c)$ neighbors with largest edge weights for each consumer node c. After initialization, each ad sorts its neighbors in parallel by descending order according to their edge weights. The sorted consumer nodes are shown in the 2 red rectangles. Each ad maintains a pointer pointing to the vertex with the largest weight of the remaining sorted neighbors.
- At the first iteration (Figure 1 (c)), each ad vertex v pours out the first $b(v)$=2 vertices from the sorted neighbors and tries to put the 2 edges into the minimum heap of the corresponding consumer vertices. However, since the capacity of each minimum heap is limited ($b(c)$=1),



(a) b-matching problem example (b) step 1: sorting neighbors (c) step 2: iteration #1 (d) step 3: iteration #2

Figure 1: An illustration of the core idea of the paralleled greedy approaches (best viewed in color).

ad vertex with the smallest edge weight will be squeezed out when the minimum heap is full. For example, in Figure 1 (c), vertex a is squeezed out from the minimum heap of vertex 2 because its weight is 6, which is smaller than the competitor vertex b's weight of 7.

- After the first iteration (Figure 1(c)), because vertex a's second neighbor with edge weight 6 is squeezed out, it moves its pointer to the next consumer and pours out 1 more consumer with index of 3, whose edge weight 4 is the largest among the remained sorted neighbors.
- After the second iteration (Figure 1(d)), all ad vertices have successfully reserved two neighbors, thus the iteration stops and the solution of the b-matching is reserved in the minimum heaps of the consumer vertices.

Intuitively the above process can be understood as a process of "pouring water". Each ad behaves like a "kettle" and each consumer behaves as a "priority-cup" (ads with smaller weights are easier to get out of the cup). Each ad iteratively pours out the sorted neighbors until the accepted vertex number equals to $b(v)$ or there are no consumers left. Finally, each pointer of the ad vertex v points to the consumer vertex whose *edge weight* is defined as the **threshold** of the weights of all neighbors. We denote this weight threshold as $\mathrm{w}_{\mathrm{thr}}(v)$. At the end of iteration, the neighbors whose edge weights are greater than $\mathrm{w}_{\mathrm{thr}}(v)$ are poured out by each vertex v. In this example, $\mathrm{w}_{\mathrm{thr}}(a) = 2$ and $\mathrm{w}_{\mathrm{thr}}(b) = 3$. Based on the analysis, the bottlenecks of the parallel greedy approaches and the way to alleviate them can be summarized as:

(1) The time complexity of the entire neighbor sorting process at step 1 is $\mathrm{O}(\delta(v)\log\delta(v))$, where $\delta(v)$ is the degree of vertex v. If we know $\mathrm{w}_{\mathrm{thr}}(v)$ for each advertiser beforehand, the sorting process of neighbors could be omitted. The reason is that we could consider $\mathrm{w}_{\mathrm{thr}}(v)$ a **pivot** (similar to the pivot in QuickSort) and only have to pour out the neighbors whose edge weights are greater than $\mathrm{w}_{\mathrm{thr}}(v)$, whose time complexity is thus reduced to $\mathrm{O}(\delta(v))$. Since $\delta(v)$ is in million scale in our case, the amount of time reduction is significant.

(2) The existing parallel greedy approaches still needs hundreds of iterations before getting the solution for large-scale problems. The reason is that each ad vertex does not know how many neighbors should be poured out beforehand. Thus it has to iteratively move its pointer until finding the right one. However, if we know $\mathrm{w}_{\mathrm{thr}}(v)$ beforehand, only one iteration is needed to produce the solution since we could pour out all neighbors whose edge weights are greater than $\mathrm{w}_{\mathrm{thr}}(v)$) once.

Once we know $\mathrm{w}_{\mathrm{thr}}(v)$ for each advertiser vertex beforehand, the time cost will be greatly reduced. In next section, we present our approach `NeuSearcher`, which can make accurate predictions of $\mathrm{w}_{\mathrm{thr}}(v)$ for new problem instances based on the historical data and compute the matching solution in a faster manner based on the estimated $\mathrm{w}_{\mathrm{thr}}(v)$.

4 Neural Searcher Framework

The proposed `NeuSearcher` is illustrated in Figure 2, which consists of two phases. (1) **Offline training**: Given a set of already solved problem instances $\mathcal{I} = \{(I^i, W^i_{\mathrm{thr}})\}$, where I^i is a solved b-matching instance and $W^i_{\mathrm{thr}} = \{\mathrm{w}_{\mathrm{thr}}(a), \forall a \in \mathbb{A}\}$ is a vector label containing a set of true weight threshold $\mathrm{w}_{\mathrm{thr}}(a)$ for all advertisers. We train a predictive model to learn the mapping from each I^i to W^i_{thr}. Specifically, a multichannel graph neural network is designed to make more accurate predictions. (2) **Online solution computing**: Given a new problem instance I^j, we utilize the already trained model to quickly predicts $\mathrm{w}_{\mathrm{thr}}(v)$ for each ad v, denoted as $\hat{W}^j_{\mathrm{thr}}$. Then, each predicted $\mathrm{w}_{\mathrm{thr}}(v)$ will be considered as a pivot value, which partitions the search space of the solution into 2 subsets. A better initial match solution could be quickly acquired within the subset with heavier edges. If all $\mathrm{w}_{\mathrm{thr}}(v)$ are correctly predicted, the initial solution is exactly the finally converged one. However, considering $\mathrm{w}_{\mathrm{thr}}(v)$ may have some deviation from the true value, we further design a parallel heuristic search model, which takes the coarse solution as input and efficiently fine-tunes it towards better qualities until convergence. Finally, we acquire the b-matching solution and the true W^j_{thr}. $(I^j, W^j_{\mathrm{thr}})$ is updated to $\mathcal{I}$, which will be further used to update the parameters of the predictive model.

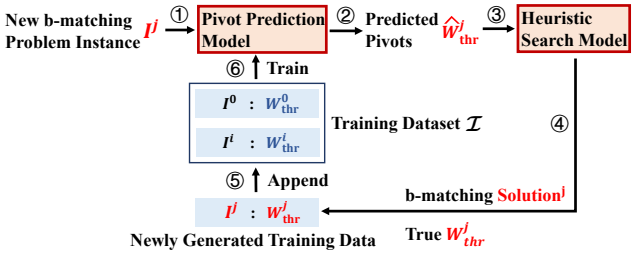


Figure 2: Neural Searcher Framework.

4.1 Pivot Prediction Model

Given a graph with node features X_v, the target is to predict $\mathrm{w}_{\mathrm{thr}}(v)$ for each ad v. To build such a predictive model, the

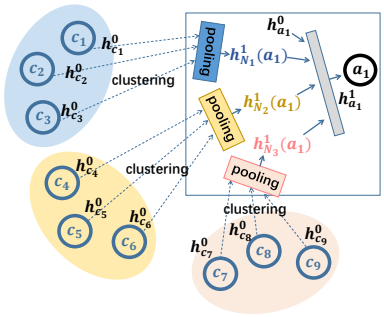


Figure 3: Illustration of our multichannel convolutional layer.

following factors should be take into consideration: (1) Since the b-matching is naturally defined in a graph, the designed model should have the capacity to capture the inherent structure (vertices, edges, constraints and their relationships) of the b-matching instances. (2) The model should be applicable to different size of graph instances and be capable of handling input dimension changes (different vertex has different number of neighbors). (3) In targeted advertising, the bipartite graphs are extremely unbalanced, i.e., $|\mathbb{C}| \gg |\mathbb{A}|$, which means the number of consumers (billions) is much larger than the number of advertisers (thousands). These considerations pose challenges to structural design of the model. In this paper, we leverage Graph Neural Networks (GNNs) [Wu *et al.*, 2019] because they could maintain the graph structure and are well-defined no matter the input graph size and the input dimension. Modern GNNs follow a neighborhood aggregation strategy, where it iteratively updates the representation of a node by aggregating representations of its neighbors. However, since the bipartite graphs are unbalanced, i.e., $|\mathbb{C}| \gg |\mathbb{A}|$, simply applying GNN with a single-channel aggregate function (e.g., even the powerful sum-pooling [Xu *et al.*, 2018]) would result in significant loss of information.

To address this issue, we design a multichannel GNN which preserves more information during aggregating and improves its representational capacity. As in Figure 3 (from an ad's view), we learn a *differentiable* soft cluster assignment matrix for nodes at each layer of a GNN, mapping candidate nodes to a set of channels. Since the learned clustering procedure assigns different nodes to different channels while putting similar nodes together, we can naturally aggregate the nodes within the same channel through sum-pooling (since they are similar) while keeping all information among different channels using concat operation (since they are different). Thus, we obtain a distribution-style summarization of the neighbors' information.

We denote the learned cluster assignment matrix at layer k as $S^{(k)} \in \mathbb{R}^{n_{a_i} \times c_k}$, where c_k is the number of channels, n_{a_i} is the number of neighboring consumers for advertiser a_i. Each row of $S^{(k)}$ corresponds to one of the n_{a_i} neighboring consumers, and each column corresponds to one of the c_k channels. Intuitively, $S^{(k)}$ provides a soft assignment of each neighboring consumer to a number of channels. Following the *aggregate* and *combine* paradigm [Wu *et al.*, 2019], Equation 4 takes the neighbor embeddings $H_{n_{a_i}}^{k-1} \in \mathbb{R}^{n_{a_i} \times d_{k-1}}$ and aggregates them according to the cluster assignments $S^{(k)}$, generating neighbors' multichannel representations $\tilde{h}_{n_{a_i}}^{k}$. Then, the multichannel representations are flattened and combined (Equation 5) with ad i's embedding $h_{a_i}^{k-1}$ at layer k-1, where $\|$ is the concat operator.

$$\tilde{h}_{n_{a_i}}^{k} = (S^{(k)^\mathsf{T}} \cdot H_{n_{a_i}}^{k-1}) \in \mathbb{R}^{c_k \times d_{k-1}} \quad \triangleright \text{AGGREGATE} \quad (4)$$

$$h_{a_i}^{k} \leftarrow \text{MLP}\left(\left[h_{a_i}^{k-1} \| \text{flatten}(\tilde{h}_{n_{a_i}}^{k})\right]\right) \quad \triangleright \text{COMBINE} \quad (5)$$

To generate the assignment matrix $S_{a_i}^{k}$ for each layer k, we apply a multi-layer perception (MLP) to the input neighbor embeddings $H_{n_{a_i}}^{k-1}$ of ad a_i, followed by a softmax layer for classification purpose:

$$S^{(k)} = \text{softmax}(\text{MLP}(H_{n_{a_i}}^{k-1})) \in \mathbb{R}^{n_{a_i} \times c_k} \quad (6)$$

where c_k is the number of clusters. After K layer aggregations, we acquire the ad node embeddings h_v^K and pass them through an MLP and finally produce a single dimension output to predict $\text{w}_{\text{thr}}(v)$ for each ad v.

$$\hat{\text{w}}_{\text{thr}}(v) = \text{MLP}(h_v^K), \forall v \in \mathbb{A} \quad (7)$$

Training Details. Taking the already solved instances $\mathcal{I} = \{(I^i, W_{\text{thr}}^i)\}$ as training data, we train the pivot prediction model end-to-end in a supervised fashion, using a mean-square error (MSE) loss. At the very beginning, when $\mathcal{I}$ is an empty set, we run the b-suitor over the recent problem instances to get the corresponding labels W_{thr}^i.

4.2 Heuristic Search Model

During the online solution computing phase when given a new b-matching problem instance, we first call the pivot prediction model trained before to predict the pivot value (i.e., the weight threshold) $\text{w}_{\text{thr}}(v)$ for each ad vertex v. Further, to ensure the solution quality, we propose a parallel heuristic search algorithm as follows. The algorithm takes the estimated pivot value $\text{w}_{\text{thr}}(v)$ as input and quickly produce a initial solution (line 4-7). Then to ensure that the b-matching solution is exactly the same with the state-of-the-art greedy approaches, a fine-tuning process (line 8-20) is developed following the idea of the parallel b-suitor algorithm. The proof is presented as follow.

The proof sketch. In [Khan *et al.*, 2016] 3.2&3.3, it proves that b-suitor achieves the same solution as serial greedy algorithm and the b-suitor finds the solution irrespective of the order of the vertices and the edges processed. Here we show that the high quality initial solution given by our method can be seen as an intermediate solution following some b-suitor processing order of vertices and edges. And since the rest fine-tuning process of Algo.1 is the same as b-suitor, our method naturally achieves exactly the same solution. Here we give the reason that our method can be seen as an intermediate solution. In Algo. 1, after the first pass of line 9-16, the solution given by our method (denoted as S) satisfies all constraints. In S, we define a set P containing all poured out edges (including all reserved and squeezed out edges). The S can be seen as an intermediate solution of b-suitor by selecting edges in P following the descending weight order from an empty solution. This completes the proof.

Algorithm 1 Parallel heuristic search algorithm

Input: Bipartite graph $G = (\mathbb{C}, \mathbb{A}, \mathbb{E})$ and a constraint function $b(v), \forall v \in \mathbb{C} \cup \mathbb{A}$, an estimated $\mathrm{w}_{\mathrm{thr}}(a), \forall a \in \mathbb{A}$. Each $c \in \mathbb{C}$ initializes a min-heap of size $b(c)$;
Output: b-matching solution;
for each vertex $a \in \mathbb{A}$ in parallel **do**
  Takes $\mathrm{w}_{\mathrm{thr}}(a)$ as the pivot and partitions the search space of all neighbors into 2 subsets;
  The heavier edges than the pivot are poured out; These edges are put into corresponding min-heaps;
  Count the number of currently reserved edges in the min-heaps. The number denoted as $\hat{b}(a)$;
end for
for each iteration **do**
  for each vertex $a \in \mathbb{A}$ in parallel **do**
    Acquires $\hat{b}(a)$, the number of reserved edge in the min-heaps currently; Denotes $b_\delta(a) = \hat{b}(a) - b(a)$;
    if $b_\delta(a) > 0$ **then**
      Recalls back $b_\delta(a)$ smallest edges preserved in the min-heaps and puts the ad vertices squeezed out by these $b_\delta(a)$ edges back into the min-heaps.
    else if $b_\delta(a) < 0$ **then**
      Pours out another $|b_\delta(a)|$ neighbors in the descending order from the remaining neighbors.
    end if
  end for
  if $b_\delta(a) == 0$ or no neighbors left, $\forall a \in \mathbb{A}$ **then**
    return edges in all min-heaps as solution;
  end if
end for

5 Experiments

5.1 Experimental Setup

Baselines. We evaluate the performance of `NeuSearcher` against the following state-of-the-art baselines. (1) ***optimal***: We use Gurobi optimizer [Gurobi, 2014] with an MIP formulation to compute the optimal solutions. (2) ***serial greedy***: The *greedy* algorithm is a practical approximate algorithm which guarantees a 1/2 approximation ratio in the worst case [Avis, 1983; Preis, 1999]. But in practical problems, its solutions are usually within 5% percent of the optimal ones [Hougardy, 2009]. (3) ***greedyMR***: [De Francisci Morales *et al.*, 2011] adapt the serial greedy algorithm to the MapReduce environment. And greedyMR is one of the fastest parallel algorithms in computing b-matching problems. (4) ***b-suitor***: *b-suitor* is the fastest (state-of-the-art) parallel approach for b-matching proposed by [Khan *et al.*, 2016]. All experiments are conducted on an Intel(R) Xeon(R) E5-2682 v4 processor based system with a memory of 128G. All codes were developed using C++ 11 multi-thread.

Datasets. We evaluate `NeuSearcher` on both open and industrial datasets. Table 1 summarizes the dataset properties. Each of the first 7 datasets (adv #1 to #7) has more than a billion edges, which are collected from the e-commerce platform of Alibaba for seven consecutive days. Due to the memory limit (128G), we cannot calculate the exact solution using Gurobi optimizer for the first 7 datasets. Thus, we compare the matching quality of the approximate algorithms relative to the exact solution on the other 3 open datasets (Amazon review data [He and McAuley, 2016] and MovieLens data [Harper and Konstan, 2016]).

Graph	# $\mathbb{C}$	# $\mathbb{A}$	# $\mathbb{E}$	Avg. Deg. of $\mathbb{A}$
adv #1 to #7	236M	46k	1B	24k
MovieLens10M	69k	10k	10M	936.6
MovieLens20M	138k	26k	20M	747.8
RatingsBooks	8M	2M	22M	9.7

Table 1: The structural properties of the datasets.

Other Settings. For the 7 advertising datasets, we use the first 4 for training, the 5th for validation and the last 2 for testing. For the other 3 open datasets, we add Gaussian noise with mean 0.0 and variance 0.1 to the edge weights and generate 4 more datasets for each (3 for training and 1 for validation). In following experiments, unless otherwise mentioned, we fix $b(v) = 0.5 * \delta(v), \forall v \in \mathbb{A}$ and set $b(v) = min\{b, \delta(v)\}, \forall v \in \mathbb{C}$, where $\delta(v)$ is the degree of v and $b = \mathrm{avg}\{\delta(v), \forall v \in \mathbb{C}\}$. For hyperparameters, we set $K = 2$, $c_k = 16$ after grid-search optimization.

Graph	serial greedy greedyMR b-suitor NeuSearcher	optimal (Gurobi)	Quality in %
MovieLens10M	29,995,076.5	**30,510,066**	99.05
MovieLens20M	60,247,629.5	**61,194,930**	98.45
RatingsBooks	77,213,078	**79,068,583**	97.65
adv #6	28,724,740.17	out-of-memory error	
adv #7	28,150,245.37	out-of-memory error	

Table 2: The solution quality comparison (best in bold).

5.2 Solution Quality Comparison

We compare the matching value of the optimal solution as well as all approximate baselines with our `NeuSearcher` in Table 2. Among the experimental results over all 5 datasets, the 4 approximation approaches, i.e., serial greedy, gredyMR, b-suitor and our NeuSearcher all find exactly the same set of matched edges with the same matching values. We summarize their results in the same column due to space limitation. Besides, in Table 2, we see that although the approximate approaches theoretically can only guarantee 1/2 approximation in the worst case, they find more than 97% of the optimal weight for the 3 open datasets. The highest approximation ratio of the approximate approaches achieved is 99.0%. For problems with larger sizes, the Gurobi fails to compute an optimal solution due to the memory limit (128G). This indicates that faster approximate approaches are good alternatives in solving large-scale b-matching problems and our `NeuSearcher` achieves the state-of-the-art solution quality.

5.3 Runtime Comparison

We provide the online solution computing time of our approach as well as runtimes of other methods over 5 datasets in Table 3. We use the same evaluation metric (clock time) to record the computing time. All results are averaged over 10 rounds. For all approaches, only CPUs are used for the

Graph	serial greedy	greedyMR	b-suitor	NeuSearcher (multichannel GNN)	optimal (Gurobi)
MovieLens 10M	92.952	32.705	35.889	**15.141**	742.795
MovieLens 20M	190.221	91.462	78.588	**35.059**	2355.614
Ratings_Books	235.607	53.387	34.627	**14.212**	44376.265 (12.3 hour)
adv #6	15352.075	1875.154	410.270	**199.423**	out-of-memory error
adv #7	14831.704	1893.876	426.359	**201.094**	out-of-memory error

Table 3: The runtimes (in seconds) of b-matching computation, where lower values are better (best in bold).

sake of fair comparison, though our model can be accelerated using GPUs. In Table 3, we see that even for the smaller open datasets, Gurobi still needs hours to compute the solutions, which is intolerable. For larger datasets adv #6 and #7, Gurobi fails and causes out-of-memory error. On the contrary, all approximate approaches are much faster than the exact algorithm. Our `NeuSearcher` with the designed multichannel GNN computes the same solutions at the fastest speed by reducing more than 50% computing time. Among other approximate methods, *b-suitor* runs faster than *greedyMR* and requires less iterations to compute the results. The serial *greedy* algorithm is the slowest since it requires a global sorting and a sequential decision process. Combining Table 2 with 3, we conclude that our `NeuSearcher` can achieve a much faster speed, while still acquire exactly the same matching solution with the state-of-the-art approaches.

5.4 Convergence Analysis

To better analyze the computing process of the three parallel approximate algorithms: greedyMR, b-suitor and our `NeuSearcher`, we plot their solution computing process in Figure 4 using adv #6 dataset as an example. We see that our approach requires the fewest (15) iterations to compute the solution. However, the b-suitor needs 68 iterations and the greedyMR needs 358 iterations. The reason is that the neural net captures the correlations between the problem structure and the weight threshold $w_{\text{thr}}(v)$ (pivot), which significantly reduces the search region of the solution space. Then, the following heuristic search algorithm benefits more from a better jumping start and only needs few steps to fine-tune the initial solution towards convergence.

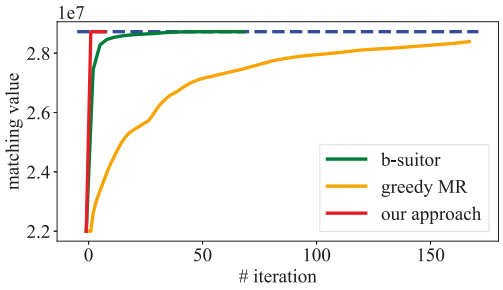
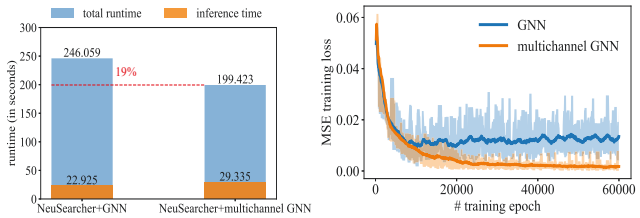


Figure 4: Matching value (of adv #6) by the greedyMR, b-suitor and our `NeuSearcher` as a function of the number of iterations.

5.5 Ablation Study: Effect of multichannel GNN

In Figure 5 (a), we compare the detailed solution computing time of NeuSearcher with multichannel GNN and NeuSearcher with GNN. We see NeuSearcher with multichannel GNN is the fastest, which reduces 19% overall computing time. Besides, we also separately compare the two inner stages of the solution computing: 1) pivot prediction (inference) and 2) fine-tuning. We see though the inference time of multichannel GNN is slightly longer than GNN, the overall time cost is much smaller, which indicates multichannel GNN provides a more precise pivot value by which reducing the subsequent fine-tuning steps. In detail, NeuSearcher with multichannel GNN only needs 15 fine-tuning iterations while NeuSearcher with GNN needs 29 iterations. Similar evidences can also be found in Figure 5 (b), where we compare the validation losses of the two models. For the reason that the multichannel GNN has a better representational ability and generalizes well, the validation loss is much lower.



(a) Training loss (b) Validation loss

Figure 5: Comparison of the runtime and validation loss of multichannel GNN and GNN in adv #6 dataset.

6 Conclusion

To the best of our knowledge, we are the first to integrate deep learning methods to accelerate solving practical large-scale b-matching problems. Our `NeuSearcher` transfers knowledge learned from previous solved instances to save more than 50% of the computing time. We also design a parallel heuristic search algorithm to ensure the solution quality exactly the same with the state-of-the-art approximation algorithms. Given highly unbalanced feature of the advertising problem, we design a multichannel graph neural network to encode the billions consumers and their diverse interests to improve the representation capability and accuracy of the pivot prediction model. Experiments on open and real-world large-scale datasets show `NeuSearcher` can compute nearly optimal solution much faster than state-of-the-art methods.

Acknowledgments

The work is supported by the Alibaba Group through Alibaba Innovative Research Program, the National Natural Science Foundation of China (Grant Nos.: 61702362, U1836214) and the new Generation of Artificial Intelligence Science and Technology Major Project of Tianjin under grant: 19ZXZNGX00010.

References

[Agrawal *et al.*, 2018] Shipra Agrawal, Morteza Zadimoghaddam, and Vahab Mirrokni. Proportional allocation: Simple, distributed, and diverse matching with high entropy. In *International Conference on Machine Learning*, pages 99–108, 2018.

[Avis, 1983] David Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13(4):475–493, 1983.

[Bayati *et al.*, 2011] Mohsen Bayati, Christian Borgs, Jennifer Chayes, and Riccardo Zecchina. Belief propagation for weighted b-matchings on arbitrary graphs and its relation to linear programs with integer solutions. *SIAM Journal on Discrete Mathematics*, 25(2):989–1011, 2011.

[Chen and Tian, 2019] Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. In *Advances in Neural Information Processing Systems*, pages 6278–6289, 2019.

[De Francisci Morales *et al.*, 2011] Gianmarco De Francisci Morales, Aristides Gionis, and Mauro Sozio. Social content matching in mapreduce. *Proceedings of the VLDB Endowment*, 4(7):460–469, 2011.

[Dhillon, 2001] Inderjit S Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 269–274. ACM, 2001.

[Ding *et al.*, 2019] Jian-Ya Ding, Chao Zhang, Lei Shen, Shengyin Li, Bing Wang, Yinghui Xu, and Le Song. Optimal solution predictions for mixed integer programs. *arXiv preprint arXiv:1906.09575*, 2019.

[Edmonds, 1965] Jack Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of research of the National Bureau of Standards B*, 69(125-130):55–56, 1965.

[Grötschel and Holland, 1985] Martin Grötschel and Olaf Holland. Solving matching problems with linear programming. *Mathematical Programming*, 33(3):243–259, 1985.

[Gurobi, 2014] Gurobi. Inc. gurobi optimizer reference manual, 2015. *URL: http://www. gurobi. com*, 2014.

[Harper and Konstan, 2016] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):19, 2016.

[He and McAuley, 2016] Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *proceedings of the 25th international conference on world wide web*, pages 507–517. International World Wide Web Conferences Steering Committee, 2016.

[He *et al.*, 2014] He He, Hal Daume III, and Jason M Eisner. Learning to search in branch and bound algorithms. In *Advances in neural information processing systems*, pages 3293–3301, 2014.

[Hougardy, 2009] Stefan Hougardy. Linear time approximation algorithms for degree constrained subgraph problems. In *Research Trends in Combinatorial Optimization*, pages 185–200. Springer, 2009.

[Khalil *et al.*, 2017] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

[Khan *et al.*, 2016] Arif Khan, Alex Pothen, Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Fredrik Manne, Mahantesh Halappanavar, and Pradeep Dubey. Efficient approximation algorithms for weighted b-matching. *SIAM Journal on Scientific Computing*, 38(5):S593–S619, 2016.

[Li *et al.*, 2018] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, pages 539–548, 2018.

[Müller-Hannemann and Schwartz, 2000] Matthias Müller-Hannemann and Alexander Schwartz. Implementing weighted b-matching algorithms: insights from a computational study. *Journal of Experimental Algorithmics (JEA)*, 5:8, 2000.

[Naim and Manne, 2018] Md Naim and Fredrik Manne. Scalable b-matching on gpus. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 637–646. IEEE, 2018.

[Padberg and Rao, 1982] Manfred W Padberg and M Ram Rao. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7(1):67–80, 1982.

[Preis, 1999] Robert Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 259–269. Springer, 1999.

[Vinyals *et al.*, 2015] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

[Wu *et al.*, 2019] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[Xu *et al.*, 2018] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[Zanfir and Sminchisescu, 2018] Andrei Zanfir and Cristian Sminchisescu. Deep learning of graph matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2684–2693, 2018.