Appendix for Towards Solving Polynomial-Objective Integer Programming with Hypergraph Neural Networks

A Details of HNN-based Framework

We present two key details of the HNN-based framework that were not covered in Section 4, allowing interested readers to reproduce our work.

A.1 Raw Features of Hypergraph Representation

We present the raw features of our hypergraph representation in Table 7. The table organizes four key components of our hypergraph representation that participate in convolutions. Each row corresponds to one component, with the first column identifying the component name, the second column listing its raw features, and the third column providing detailed descriptions of these features. Specifically, the variable vertices $\mathcal{V}$ are assigned nine-dimensional raw features that encode variable types, bound information, and their roles in the objective function. Constraint vertices $\mathcal{C}$ are assigned four-dimensional raw features based on their constraint sense and right-hand-side values. Hyperedges $\mathcal{H}$ are assigned raw features whose length varies according to the number of variables they contain, as introduced in Section 4.1. For a variable v contained in a hyperedge ϵ, a $\omega_{v\epsilon}$ containing the term coefficient and the variable's exponent is added to ϵ's raw features. Finally, standard edges $\mathcal{E}$ are assigned two-dimensional features that reflect coefficients and degrees of the corresponding variables within their associated constraints.

A.2 Neighborhood Search for Repair-and-Refinement

We implement parallel neighborhood optimization as described in [6, 5], which incorporates two key components: a Q-repair-based repair strategy that efficiently repairs model predictions into feasible solutions, and an iterated multi-neighborhood search that refines these solutions to achieve higher quality. In the following, we provide detailed descriptions of both components.

Q-Repair-Based Repair Strategy The Q-repair begins by selecting the αn variables with the largest predicted loss values to optimize, while fixing the remaining $(1-\alpha)n$ variables to their predicted values. Here, $\alpha \in [0, 1]$ is a proportion that determines the neighborhood search size and n represents the total number of variables. Then Q-repair traverses constraints to identify those that cannot be satisfied. This identification follows a greedy approach: calculating

Table 7: Raw Features of High-Degree Term-Aware Hypergraph Representation

Tensor	Feature	Description
$\mathcal{V}$	type	(continuous, binary, integer) as a one-hot encoding
	lb	Lower bound value of the variable
	up	Upper bound value of the variable
	inf_lb	Binary indicator (1 if the lower bound is negative infinity, 0 otherwise)
	inf_ub	Binary indicator (1 if the upper bound is positive infinity, 0 otherwise)
	avg_obj_coe	Average value of coefficients associated with this variable in the objective function
	avg_obj_deg	Average degree of this variable across all terms in the objective function
$\mathcal{C}$	sense	($<$, $>$, $=$) as a one-hot encoding
	rhs	Numerical value on the right-hand side of the constraint
$\omega_{v\epsilon}$	deg	Degree of each variable in the high-degree term
	coe	Coefficient value associated with the high-degree term
$\mathcal{E}$	avg_coe	Average value of coefficients across all terms containing the variable in the associated constraint
	avg_deg	Average degree of this variable across all terms containing it in the associated constraint

the upper and lower bounds of each term in the left-hand side, summing these bounds, and comparing the result with the right-hand side. When an unsatisfied constraint is detected, the variables involved in this constraint are incrementally added to the neighborhood until either all variables from that constraint have been incorporated or the neighborhood reaches a size limit of $\alpha_{\text{ub}}n$ variables. Q-repair terminates after evaluating all constraints and returns the neighborhood (i.e., variables to be optimized) for repair.

Subsequently, the repair strategy employs exact solvers (such as Gurobi and SCIP) to optimize the subproblem defined by the Q-repair neighborhood. If no feasible solution is identified within the allocated time, Q-repair is repeated with an enlarged initial $\alpha = \alpha_{\text{step}} + \text{len}(\text{neighborhood})/n$, followed by another neighborhood search on the new expanded neighborhood. This iterative process continues until a feasible solution is found, or α exceeds 1, or the maximum time to repair-and-refine has been reached.

Iterated Multi-Neighborhood Search The iterated multi-neighborhood search begins by generating a set of initial neighborhoods using a sequential filling approach. Specifically, this process first randomly shuffles all constraints. Then, it iteratively processes each constraint by sequentially adding its variables to the current neighborhood. When the predefined neighborhood size limit is reached, a new neighborhood is created and the process continues, until all constraints

and their associated variables have been assigned to neighborhoods. This process creates multiple neighborhoods where variables from the same constraint tend to appear together in the same neighborhoods, thereby reducing the likelihood of constraint violations. Next, using the solution obtained by the repair strategy as a starting point, subproblems are formulated based on each neighborhood and optimized using exact solvers.

After that, the algorithm generates crossover neighborhoods to explore combinations of different subproblem solutions. It groups all neighborhoods into pairs. For two neighborhoods N_1 and N_2 in a pair with their respective subproblem solutions $x^{(1)}, x^{(2)}$, assuming $x^{(1)}$ has equal or better objective value than $x^{(2)}$, a crossover neighborhood is created through two steps: 1) constructing a crossover solution x' by taking $x'_i = x_i^{(1)}$ for variables in N_1 and $x'_i = x_i^{(2)}$ for other variables, and 2) applying Q-repair on x'. Then, subproblems based on these crossover neighborhoods are optimized. The algorithm selects the best solution among all the candidates, both initial neighborhoods and crossover neighborhoods, to serve as the starting point for the next iteration. These two processes repeat until the predetermined time limit is reached, with the best solution found across all iterations returned as the final result.

B Details of Benchmarks

This section introduces the details of the synthetic datasets used in our experiments.

B.1 Details of Synthetic Quadratic Instances

In Section 5.3, we evaluate the efficiency of our HNN-based framework using two synthetic quadratic datasets: QMKP and RandQCP, which are generated and provided by [5]. The formulations of these problems are presented below.

The Quadratic Multiple Knapsack Problem (QMKP) extends the classic knapsack problem by incorporating multiple weight constraints and quadratic profit terms. It involves selecting items to place in a knapsack with limited capacity across multiple weight dimensions. Each item yields an individual profit, while specific pairs of items generate additional interactive profits when selected together. The objective is to maximize the total profit while adhering to capacity constraints. QMKP can be formulated as a quadratic programming problem as shown in Eq. 9-11:

$$\max \quad \sum_i c_i x_i + \sum_{(i,j)\in E} q_{ij} x_i x_j, \tag{9}$$

$$\text{s.t.} \quad a_i^k x_i \le b^k, \quad \forall k \in M, \tag{10}$$

$$x_i \in \{0,1\}, \quad \forall i \in N, \tag{11}$$

where x_i is a binary variable indicating whether item i is selected, c_i represents the individual profit for item i, and q_{ij} denotes the interactive profit obtained

by selecting both items i and j. The set E contains item pairs with interactive profits, a_i^k represents the k-th weight of item i, and b^k denotes the knapsack's capacity on the k-th weight dimension. M and N represent the total number of weight dimensions and items, respectively.

The Random Quadratically Constrained Quadratic Program (RandQCP) is an extension of the independent set problem. It aims to select vertices from a hypergraph to maximize total weights while satisfying specified constraints on each hyperedge. The quadratic programming formulation of RandQCP is given in Eq. 12-14.

$$\max \quad \sum_{i \in V} c_i x_i, \tag{12}$$

$$\text{s.t.} \quad \sum_{i \in e} a_i x_i + \sum_{i,j \in e, i \neq j} q_{ij} x_i x_j - |e| \leq 0, \quad \forall e \in \mathcal{E}, \tag{13}$$

$$x_i \in \{0, 1\}, \quad \forall i \in V, \tag{14}$$

where V represents the set of vertices, $\mathcal{E}$ denotes the hyperedge set, c_i is the weight associated with vertex i, and a_i and q_{ij} are the limitation coefficients for selecting vertex i and vertex pair (i, j), respectively. The term e refers to a specific hyperedge, and $|e|$ indicates the number of vertices contained within hyperedge e.

For details of generation and access to the generated datasets, please refer to the original paper by [5].

B.2 Details of Synthetic Quintic Instances

To evaluate the effectiveness of our HNN-based method on more complex integer programming problems, we generated synthetic quintic datasets based on the Capacitated Facility Location Problem under Traffic Congestion (CFLPTC) inspired by [1] and [3]. The formulation and generation procedures are detailed below.

Formulation of CFLPTC CFLPTC extends the standard capacitated facility location problem by incorporating traffic congestion effects. Consider a scenario with m customers $J = \{1, \cdots, m\}$ and n potential facility locations $I = \{1, \cdots, n\}$. Each customer j has a demand D_j, while each facility at location i incurs an opening cost o_i and has a capacity C_i. Once opened, a facility can serve customers provided that the total demand it satisfies does not exceed its capacity. Each customer must be served by exactly one opened facility. The transportation cost for serving customer j from facility i depends on the distance between them d_{ij} and the traffic congestion level. The objective is to determine which facilities to open and how to assign customers to these facilities, so that the total cost comprising facility opening costs and transportation expenses is minimized. The mathematical formulation is presented in Eq. 15-20.

$$\min \quad \sum_{i \in I} o_i y_i + \sum_{i \in I} \sum_{j \in J} \alpha(1 + 0.15 e_i^{\beta}) d_{ij} x_{ij} \tag{15}$$

$$\text{s.t.} \quad \sum_{i} x_{ij} = 1, \forall j \in J, \tag{16}$$

$$x_{ij} \leq y_i, \forall i \in I, j \in J, \tag{17}$$

$$\sum_{j} D_j x_{ij} \leq C_i y_i, \forall i \in I, \tag{18}$$

$$e_i = \frac{\sum_j D_j x_{ij} + b_i}{T_i}, \forall i \in I, \tag{19}$$

$$x_{ij}, y_i \in \{0, 1\}, \forall i \in I, j \in J. \tag{20}$$

where y_i and x_{ij} are binary variables to determine whether to open the facility at location i and whether to assign customer j to the facility at location i, separately.

In the objective function Eq. 15, the transportation cost from facility i to customer j is expressed as $\alpha(1 + 0.15 e_i^{\beta}) d_{ij} x_{ij}$, where the term $\alpha(1 + 0.15 e_i^{\beta})$ quantifies the additional cost induced by traffic congestion. This formulation, together with Eq. 19 which determines e_i, is derived from the Bureau of Public Roads (BPR) function, an empirical formula for estimating increased transportation time corresponding to congestion level [4]. In this context, T_i represents the total traffic capacity surrounding facility location i and b_i denotes the background traffic flow in the vicinity. The parameters α and β are typically set to 1 and 4 respectively, which make CFLPTC a quintic programming problem.

While CFLPTC technically falls under the category of mixed-integer programming due to its combination of binary variables (x_{ij}, y_i) and continuous variables (e_i), it remains essentially an integer programming problem. This is because the continuous variables e_i are merely auxiliary and completely determined by the binary assignment variables x_{ij}. Therefore, it is methodologically reasonable to include CFLPTC as a dataset in this work, which focuses on integer programming problems.

Quadratic Reformulation of CFLPTC In Section 5.1, we compared our method against NeuralQP on the quintic CFLPTC instances. However, NeuralQP is designed exclusively for quadratic optimization problems and cannot directly handle the quintic terms present in the original CFLPTC formulation. To enable this comparison, we reformulated the quintic CFLPTC instances into equivalent quadratic problems by introducing auxiliary variables that decompose higher-order terms. The reformulation strategy systematically replaces quintic terms with chains of quadratic relationships. Specifically, for each $i \in I$, we define $e_{1i} = e_i^2$ and $e_{2i} = e_{1i}^2$, which transform the quintic terms $e_i^4 x_{ij}$ into quadratic terms $e_{2i} x_{ij}$. The complete quadratic reformulation is presented in Eq. 21-24.

$$\min \quad \sum_{i\in I} o_i y_i + \sum_{i\in I}\sum_{j\in J} \alpha(1+0.15e_{2i})d_{ij}x_{ij} \tag{21}$$

$$\text{s.t.} \quad \text{Eq. 16 - 20}, \tag{22}$$

$$e_{1i} = e_i^2, \forall i \in I, \tag{23}$$

$$e_{2i} = e_{1i}^2, \forall i \in I, \tag{24}$$

It is important to note that while lower-degree objective functions and constraints are generally more tractable for optimization algorithms than their higher-degree counterparts, the reformulation process inevitably introduces additional variables and constraints that can impose significant computational overhead. For CFLPTC instances, the quadratic reformulation requires $2n$ additional variables (e_{1i}, e_{2i}) and $2n$ additional quadratic constraints (Eq. 23 and 24), substantially increasing the complexity. The increase of complexity may offset or even outweigh the computational benefits gained from degree reduction, as solvers must now handle a larger search space and a more complicated constraint set. Consequently, reformulating high-degree problems into lower-degree equivalents does not guarantee improved optimization efficiency; the net effect depends on the trade-off between reduced degree and increased problem complexity, which varies with specific problem characteristics and solver capabilities. This trade-off underscores the importance of developing optimization methods that can directly handle high-degree integer programming problems rather than relying solely on quadratic reformulations.

Instance Generation Following the approach in [3], we generated datasets at four distinct scales for training, as detailed in Table 8. The notation $U(a, b)$ indicates that the corresponding parameters are randomly sampled from a uniform distribution ranging from a to b (inclusive). Both customer and facility locations were generated within a two-dimensional Euclidean space according to the "Coordinate" specifications in Table 8, with distances calculated using the Euclidean metric. Consistently across all datasets, the total traffic capacity T_i was generated as $U(1, 4) \cdot C_i$, while the background traffic flow b_i was set to $U(0.1, 1) \cdot T_i$.

Table 8: Setting for CFLPTC Training Dataset Generation

Dataset	Number	m	n	Coordinate	D_j	o_i	C_i
1	1605	50	10	$U(10,200)$	$U(10,50)$	$U(300,700)$	$U(100,500)$
2	1119	50	20	$U(10,200)$	$U(30,80)$	$U(300,700)$	$U(100,500)$
3	984	150	30	$U(10,300)$	$U(10,50)$	$U(300,700)$	$U(200,600)$
4	200	200	30	$U(10,200)$	$U(10,50)$	$U(500,1500)$	$U(500,800)$

For testing purposes, we generated 16 instances each at the 150×30 scale and the 200×30 scale, adhering to the same parameter settings used for training datasets 3 and 4, respectively. Additionally, we created 10 larger instances at the 500×100 scale, following the parameter settings of training dataset 1 but with adjusted values for m and n. These testing datasets enable comprehensive evaluation of our model's capability to effectively tackle complex, large-scale integer programming problems with high-degree terms.

C Implementation Details

Model Details First, all raw features of the input hypergraph were transformed into initial embeddings through 2-layer MLPs activated by LeakyReLU, where the dimensions of hidden spaces and output features are 64 and 16, respectively. The number of iterations for executing hyperedge-based convolution is $L = 6$. The negative slopes of all LeakyReLU activations are set to 0.1.

Training Details We utilized AdamW with a learning rate of 1e-4 and weight decay of 1e-4 as the optimizer to train our model. We set the batch size to 64 and training epochs to 100. On each training dataset, our HNN models were trained on a supercomputer node with an NVIDIA A100 GPU and an 18-core Intel Xeon Platinum 8360Y CPU. For fair comparison, we used the same device to train the models of learning-based baselines, with the same hyper-parameter settings as in their original papers.

Inference Details We used Gurobi 12.0.0 and SCIP 9.2.0 for all inference tests, which were run exclusively on CPUs. Tests using SCIP were conducted on a supercomputer equipped with an AMD Rome 7H12 CPU, while those using Gurobi were run on a separate supercomputer with an Intel Xeon Platinum 8260 CPU. Note that this setup does not introduce unfairness, as our comparisons focus on the performance of different methods within the same exact solver, rather than comparing the solvers themselves.

We implemented the repair-and-refinement algorithm (see Appendix A.2) following the parameter settings proposed by [5]. Specifically, for the Q-repair-based repair strategy, we initialized the parameter α at 0.1, with $\alpha_{\text{ub}} = 1$ and $\alpha_{\text{step}} = 0.05$. For the iterated multi-neighborhood search, the neighborhood size is defined as half the number of problem variables. For each subproblem occurring in both the Q-repair-based repair strategy and the iterated multi-neighborhood search, we set a maximum wall-clock time of 60 seconds when addressing largest-scale instances: 10k-scale QMKP and RandQCP problems, and 500×100-scale CFLPTC datasets. All other testing datasets were limited to 30 seconds per subproblem. The repair-and-refinement stops when the total wall-clock time reaches the preset limit (see Section 5.1).

Details of the Ablation Baselines In the ablation studies (Section 5.4), we constructed two ablation baselines (w/o-HyConv and w/o-VCConv) to investigate

the contributions of hyperedge-based convolution and variable-constraint-based convolution, as well as two additional ablation baselines (NeuralQP-HD and GNNQP-HD) to examine the role of our hyperedge-based convolution in parsing high-order relationships from high-degree terms. The first two ablation baselines are constructed to remain as comparable as possible to our HNN model while omitting the targeted convolution modules. Since simply removing a component would disable the model from capturing one key relationship in IPHD, we make slight but necessary adjustments to their input representations. For w/o-HyConv, the only change is the removal of hyperedges from the representation. For w/o-VCConv, its hypergraph representation contains the same variable and constraint vertices as in our representation but differs in that it has no edges and uses alternative hyperedges. These hyperedges encode both variable interactions in high-degree terms and variable-constraint interdependencies: each term is represented by a hyperedge connecting its variables and the constraint to which it belongs. The hyperedge features follow the same design as our representation.

For NeuralQP-HD and GNNQP-HD, we replace the hyperedge-based convolution of our model with the convolutions for high-degree terms from NeuralQP [5] and GNNQP [2], respectively. Specifically, NeuralQP-HD adds additional vertices to its hypergraph representation to represent degrees, and each of its hyperedges connects variable vertices and degree vertices if one high-degree term contains the corresponding variables with the exponents of the corresponding degrees. It also applies a two-step convolution similar to our Eq. 4 and Eq. 5 to capture high-order relationships. GNNQP-HD uses hyperedges to connect variable vertices if they appear in the same high-degree term, and it allows a variable vertex to appear k times in the hyperedge if its variable's degree is k in this term. To pass information from high-order relationships, it also uses convolution layers to aggregate hyperedges into variable embeddings. Other model structures, including the embedding initialization, the variable-constraint convolution, and the output layer, all remain the same as our model for fairness.

D Additional Experiments to Evaluate Model Prediction

In Section 5 we have demonstrated the effectiveness of the complete HNN-based framework composed of both HNN prediction and repair-and-refinement. To assess the quality of our HNN model's predictions as initial solution values without refinement, in this section we conducted additional experiments that isolate the model's predictive performance from the overall framework. We applied our HNN models trained on RandQCP's training data to the RandQCP test sets with 10k-scaled instances, and models trained on QMKP's training data to the QMKP test sets with 10k-scaled instances. These largest-scale testing datasets are selected to rigorously assess prediction performance for challenging instances. Since our HNN model generates initial solution values rather than directly producing feasible solutions, we applied the Q-Repair-Based Repair Strategy based on Gurobi (detailed in Appendix A.2) to convert model predictions into feasible solutions, with no further refinement performed. We compared against Neu-

ralQP with identical settings and Gurobi configured to prioritize finding feasible solutions by setting "Params.MIPFocus = 1", "Params.NonConvex = 2", and "Params.SolutionLimit = 1".

We evaluated performance using three comprehensive metrics listed below, and present the comparative results in Table 9.

- Feasible ratio: The percentage of model predictions that yield feasible solutions before repair. A higher feasible ratio indicates stronger constraint satisfaction capability.
- $\text{gap}_{\%}$: introduced in Section 5.1.
- Wall-clock time: For our method and NeuralQP, it is the time required to obtain a feasible solution through the repair process, while for Gurobi it is the time required to obtain the first feasible solution. Shorter times indicate that the model's predictions can be more efficiently converted into feasible solutions.

The results in Table 9 demonstrate that our HNN model achieves superior solution quality, as evidenced by consistently lower mean $\text{gap}_{\%}$ values compared to both baselines. This indicates that our model's predictions, after repair, are closer to the best-known solutions and provide higher-quality initial solution values for optimization.

Table 9 also exhibits that our method shows a lower feasible ratio before repair and longer repair times compared to the baseline methods. While these metrics might initially suggest limitations, a closer examination reveals that they do not represent true disadvantages. In terms of feasible ratio, although NeuralQP achieved a higher feasible ratio, both NeuralQP and Gurobi frequently generated trivial solutions with all variables set to zero. Such trivial solutions, while technically feasible, provide less guidance for subsequent refinement processes. Regarding computational time, although our method requires longer repair times than NeuralQP and Gurobi, the actual repair time remains very short (less than 1 second), which is highly acceptable given that 10,000-variable instances typically require extensive search times. In summary, the comparative results demonstrate that our HNN model is a practical choice for generating high-quality initial solution values.

Table 9: Comparison of our HNN model, NeuralQP and Gurobi in terms of prediction performance.

Method	QMKP			RandQCP		
	feasible ratio (%)	$\text{gap}_{\%}$	time (ms)	feasible ratio (%)	$\text{gap}_{\%}$	time (ms)
Gurobi	–	100	5.30	–	100	2.49
NeuralQP	100	99.10	163	0	53.30	392
Ours	66.67	77.40	946	0	51.74	835

E Complexity Analysis

This section analyzes the memory requirements of the proposed hypergraph representation and the arithmetic time complexity of the proposed HNN's inference. We consider an IPHD instance with n variables, m constraints, and n_h high-degree terms. Let s denote the total number of variable occurrences across all high-degree terms, and let n_e denote the total number of variable-constraint incidences, i.e., the number of times any variable appears with a nonzero coefficient in any constraint. These parameters allow us to demonstrate the efficiency of our method in terms of both memory usage and computational complexity, as shown in the following subsections.

E.1 Memory Requirement for the Hypergraph Representation

According to Section 4.1 and Appendix A.1, hypergraph representation of the IPHD instance comprises four components:

- n variable vertices, each with 9 raw features;
- m constraint vertices, each with 4 features;
- n_h hyperedges, with s vertex-hyperedge coefficients, where each coefficient contains 2 floats;
- n_e edges, each with 2 features;

Variable vertices and constraint vertices can be stored using their indices, while hyperedges and edges can be stored using tuples of vertex indices they contain. In total, hypergraph structure requires $(n + m + s + 2n_e)$ indices to represent. Additionally, there are $(9n + 4m + 2n_e + 2s)$ raw features. Assuming all indices are stored as 4-byte integers and raw features are stored as 8-byte floats (double precision), the total memory requirement for the hypergraph representation is:

$$\textbf{bytes} = 76n + 36m + 20s + 24n_e. \tag{25}$$

To illustrate this with a concrete example, consider the largest CFLPTC instances we tested, which involve 500 customers and 100 facilities. As detailed in Section B.2.1, these instances have $n = 50,200, m = 50,700, n_e = 200,300, s = 100,000$. Applying Eq. 25, the total memory requirement is 12,447,600 bytes, or approximately 11.87 megabytes (MB). This represents a very manageable memory overhead for modern hardware, demonstrating that our hypergraph representation remains practical even for large-scale instances.

E.2 Arithmetic Time Complexity for the HNN

In this subsection, we analyze the arithmetic complexity of our HNN model during inference. Let n_{hid} denote the largest dimension among raw features, hidden embeddings, and outputs, and assume we perform L_{hyper} hypergraph-based convolutions and L_{bi} bipartite-graph-based convolutions. The complexity analysis for each component is as follows:

– Initial embedding: it is a 2-layer MLP applied on all raw features, with arithmetic complexity $O((n+m+s+n_e)n_{\text{hid}}^2)$;
– Hypergraph-based convolution:
  • Eq. 4 performs weighted summation with complexity $O(sn_{\text{hid}})$;
  • Eq.5 combines weighted means, a 2-layer MLP, and a residual connection, with complexity $O(sn_{\text{hid}})$, $O(nn_{\text{hid}}^2)$, and $O(nn_{\text{hid}})$, separately. The total complexity is $O(sn_{\text{hid}} + nn_{\text{hid}}^2)$;
  • Overall complexity: $O(L_{\text{hyper}}(sn_{\text{hid}} + nn_{\text{hid}}^2))$;
– Bipartite-graph-based convolution:
  • Eq. 6 combines summations, a 2-layer MLP, and residual connection, with complexity $O(n_e n_{\text{hid}}$, $O(mn_{\text{hid}}^2)$, and $O(mn_{\text{hid}})$, separately. The total complexity is $O(n_e n_{\text{hid}} + mn_{\text{hid}}^2)$;
  • Eq. 7 has similar structure to Eq. 6, with complexity $O(n_e n_{\text{hid}} + nn_{\text{hid}}^2)$;
  • Overall complexity: $O(L_{\text{bi}}(n_e n_{\text{hid}} + mn_{\text{hid}}^2 + nn_{\text{hid}}^2))$;
– Output layer: A 2-layer MLP applied to variable embeddings, with complexity $O(nn_{\text{hid}}^2)$.

Therefore, the overall arithmetic complexity of HNN inference is $O(n_{\text{hid}}(L_{\text{hyper}}s + L_{\text{bi}}n_e) + n_{\text{hid}}^2(L_{\text{hyper}}n + L_{\text{bi}}n + L_{\text{bi}}m))$. Since n_{hid}, L_{hyper}, and L_{bi} are fixed constants in our experiments (see Section 5.2), the arithmetic complexity simplifies to $O(n+m+s+n_e)$, which scales linearly with the number of variables, constraints, hyperedge density, and edge density.

Hypergraph representations for integer programming problems are typically sparse in both hyperedges and edges, making our HNN model highly efficient. To demonstrate robustness, we consider the extreme case of a fully dense hypergraph representation where every pair of variable and constraint vertices is connected by edges, and all variable vertices are connected within each hyperedge. In this scenario, $s = n_h n$ and $n_e = nm$, yielding a quadratic complexity $O(n(m + n_e))$. This analysis shows that even in such extreme cases, which rarely occur in practice, our HNN model maintains good computational efficiency for inference.

Bibliography

[1] Bai, Y., Hwang, T., Kang, S., Ouyang, Y.: Biofuel refinery location and supply chain planning under traffic congestion. Transportation Research Part B: Methodological **45**(1), 162–175 (2011)

[2] Chen, Z., Chen, X., Liu, J., Wang, X., Yin, W.: Expressive power of graph neural networks for (mixed-integer) quadratic programs. In: ICML (2025)

[3] Holmberg, K., Rönnqvist, M., Yuan, D.: An exact algorithm for the capacitated facility location problems with single sourcing. European Journal of Operational Research **113**(3), 544–559 (1999)

[4] United States Bureau of Public Roads: Traffic Assignment Manual for Application with a Large, High Speed Computer, vol. 37 (1964)

[5] Xiong, Z., Zong, F., Ye, H., Xu, H.: NeuralQP: A general hypergraph-based optimization framework for large-scale QCQPs. ArXiv preprint **abs/2410.03720** (2024)
[6] Ye, H., Xu, H., Wang, H., Wang, C., Jiang, Y.: Gnn&gbdt-guided fast optimizing framework for large-scale integer programming. In: ICML. Proceedings of Machine Learning Research, vol. 202, pp. 39864–39878 (2023)