Generating Realistic and Difficult Job Shop Scheduling Problems with a GAN-based Framework for Graph Neural Networks

Pablo Ariño Fernández[1]

***Abstract*— The complexity of Job Shop Scheduling Problems (JSSPs) in real-world scenarios often surpasses the capabilities of traditional instance generation methods, necessitating a more sophisticated approach for effective testing and benchmarking of scheduling algorithms. This paper introduces a novel framework leveraging adjacency-matrix-based Generative Adversarial Networks (GANs) for the generation of realistic and challenging problems represented as a disjunctive graph. Furthermore, we Empirical experiments show that predicting effectively the difficulty of a problem is a challenging task, and that future work is needed to address this issue. Our source code is publicly available at `https://github.com/Pabloo22/gnn_scheduler`.**

1 Introduction

The Job Shop Scheduling Problem (JSSP) represents an extensively studied aspect of scheduling research, attracting significant attention from scholars in both engineering and academic disciplines [1]. This type of scheduling challenge focuses on optimally assigning machines to various jobs, each involving a series of operations. This must be done while adhering to certain constraints like processing precedence and machine-sharing. The JSSP is of keen interest in research due to its applicability in devising effective manufacturing system schedules and its role as a demanding benchmark for NP-hard combinatorial optimization problems [2], aiding in the development of combinatorial optimization algorithms.

In a classical JSSP setting, we have a job shop environment with a set of machines $M = \{M_1, M_2, \ldots, M_m\}$ and jobs $J = \{J_1, J_2, \ldots, J_i, \ldots, J_n\}$, where each job J_i consists of a sequence of operations $O_i = \{O_{i1}, O_{i2}, \ldots, O_{ij}, \ldots, O_{in_i}\}$. These operations must be processed in a specified order. Each operation O_{ij} is assigned to a machine in M, with a designated processing time or duration d_{ij}. Additionally, for each machine M_i, we define M_i as a set containing all operations assigned to machine i. This set captures the variety of operations that a particular machine is responsible for, without implying any specific processing order. The primary goal in JSSP is to sequence the operations across all machines to minimize the maximum completion time of all jobs, also known as the makespan. In this work, we limit ourselves to basic problems: the processing times are constant, and there are no set-up times, due dates, or release dates, among other factors.

Traditionally in the field of Combinatorial Optimization (CO), the focus has been on finding optimal solutions [3]; utilizing heuristics for quicker, good-enough solutions [4]; or developing approximate solutions when exact ones are unfeasible [5]. Recently, the emphasis has shifted towards data-dependent and machine learning approaches, especially for problems with common patterns or characteristics [6, 7]. In particular, Graph Neural Networks (GNNs) have emerged as a powerful tool, exploiting graph-based representations of problems to enhance understanding and solution efficiency [8]. JSSPs can be represented as disjunctive graphs [9]. This representation does not assume any order for jobs and machines, and problems of any number of jobs and machines can be successfully handled with GNNs. Some examples which use a GNN-based approach for solving JSSPs are [10, 11].

On the other hand, generating challenging and realistic instances has been an important problem for creating benchmarks which allow a quality comparison of different algorithms. Previous methods for generating hard-to-solve instances with a set of desired properties rely primarily on generating a large number of instances and selecting those that are most difficult based on certain criteria, such as the divergence of the solution from a lower bound or the variability of solutions obtained from different starting points [12, 13].

These traditional instance generation methods, however, fall short when it comes to training machine learning algorithms. The central issue with these methods is their inefficiency: they require generating a large number of instances and then individually assessing each to identify their specific properties. This process becomes particularly burdensome when the desired properties are seldom found in randomly generated instances or are costly to evaluate. Consequently, such methods could be impractical for the extensive data needs of machine learning.

In this work, we propose to use a modification of the adjacency-matrix-based Generative Adversarial Network (GAN) [14] proposed in [15] for generating molecular graphs. By *learning* a generative model from the data, we can generate synthetic data on the fly with a desired set of properties for training new machine learning models. To the best of our knowledge, this is the first project that uses a deep generative model for generating novel JSSP instances.

2 Background

2.1 Job Shop Scheduling Problems as Graphs

Every JSSP can be modeled by a disjunctive graph [9]. The disjunctive graph is created by first adding nodes

[1]Polytechnic University of Madrid. Email: pablo.arino@alumnos.upm.es

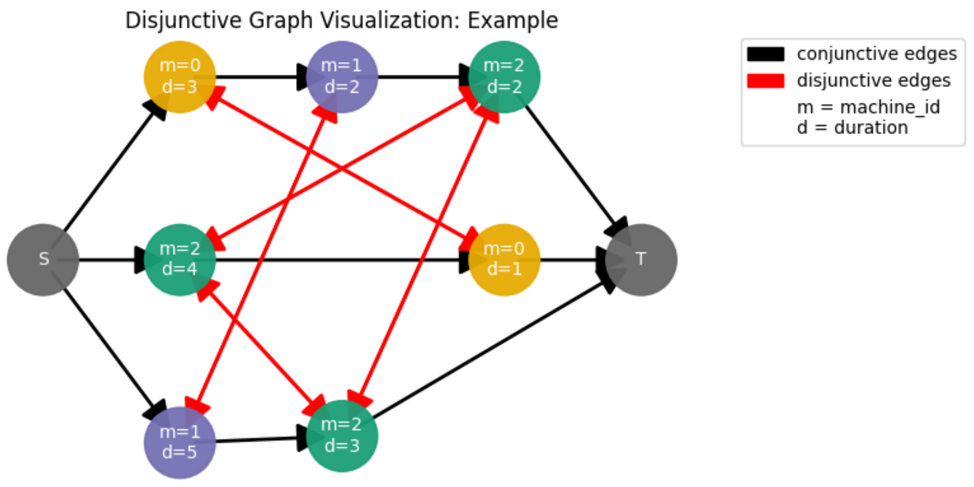


Fig. 1: Example of a Disjunctive Graph.

representing each operation in the jobs, along with two special nodes: a source S and a sink T. Each operation node is linked to the next operation in its job sequence by conjunctive edges, forming a path from the source to the sink. These edges represent the order in which operations of a single job must be performed.

Additionally, the graph includes disjunctive edges between operations that use the same machine but belong to different jobs. These edges are bidirectional, indicating that either of the connected operations can be performed first. See Figure 1 for an example of a problem defined as follows.
Given:

- Machines $M = \{M_0, M_1, M_2\}$
- Jobs $J = \{J_1, J_2, J_3\}$

The operations for each job are:

$$J_1 : O_{11}(M_0, 3 \text{ units}), O_{12}(M_1, 2 \text{ units}), O_{13}(M_2, 2 \text{ units})$$
$$J_2 : O_{21}(M_2, 4 \text{ units}), O_{22}(M_0, 1 \text{ unit})$$
$$J_3 : O_{31}(M_1, 5 \text{ units}), O_{32}(M_2, 3 \text{ units})$$

2.2 Relational Graph Convolutional Networks

Relational Graph Convolutional Networks (R-GCNs) [16] extend the capabilities of traditional Graph Convolutional Networks (GCNs) [17] to handle different types of relations in graphs.

An R-GCN is constructed by defining a set of relation-specific transformation matrices. For a graph with L different types of relations, the R-GCN layer is formulated as:

$$h_i^{(l+1)} = \sigma\left(\sum_{r\in R}\sum_{j\in N_i^r} \frac{1}{c_{i,r}} W_r^{(l)} h_j^{(l)} + W_0^{(l)} h_i^{(l)}\right) \quad (1)$$

where $h_i^{(l+1)}$ is the feature vector of node i in layer $l+1$, σ is a non-linear activation function. The term R represents different relation types, N_i^r is the set of neighbor indices of node i under relation r,and $c_{i,r}$ is a normalization constant. The weights $W_r^{(l)}$ and $W_0^{(l)}$ are learnable weight matrices for each relation type r and self-connections, respectively.

In the context of JSSP, R-GCNs can effectively capture the complex dependencies between different operations across various machines and jobs. By treating different types of edges (conjunctive and disjunctive) in the disjunctive graph as separate relations.

2.3 Implicit vs. Likelihood-Based Methods

While likelihood-based methods like the Variational Autoencoder (VAE) typically offer easier and more stable optimization [18, 19], they pose significant challenges in graph-structured data generation, such as the Job Shop Scheduling Problems (JSSPs). For JSSPs represented as graphs, these methods would require expensive graph matching procedures or explicit evaluation of likelihood for all node permutations [20]. This complexity arises due to the need for invariance to the reordering of nodes in the graph's matrix representation.

In contrast, implicit generative models, particularly Generative Adversarial Networks (GANs), provide a compelling alternative. GANs, introduced by Goodfellow et al. [14], operate without the need for an explicit likelihood function. This framework includes a generative model G_θ and a discriminative model D_ϕ, both implemented as neural networks and trained simultaneously. The generator G_θ learns to map from a prior noise distribution to the data distribution, generating new data points, while the discriminator D_ϕ learns to differentiate between real and generated samples.

Their training process can be described as a minimax game, expressed as

$$\min_\theta \max_\phi \Big[\mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D_\phi(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D_\phi(G_\theta(z)))]\Big]. \quad (2)$$

Here, the generator aims to produce samples that deceive the discriminator, while the discriminator strives to accurately distinguish between real and generated samples.

For JSSP graph generation, using GANs allows the model to bypass the need for node ordering invariance. The generator, despite having to decide on a specific node ordering, is not constrained by a likelihood evaluation, giving it freedom to choose any suitable ordering. This flexibility is crucial in generating diverse and challenging JSSP instances.

To enhance training stability and prevent issues like mode collapse, our framework incorporates techniques such as minibatch discrimination and the use of an improved Wasserstein GAN (WGAN) model [21, 22].

2.3.1 Improved WGAN

The Improved Wasserstein GAN (WGAN) presents a refinement over the original WGAN framework [22, 23]. WGANs aim to minimize an approximation of the Earth Mover (EM) distance, also known as the Wasserstein-1 distance, between two probability distributions. The Wasserstein distance offers a more stable training of GANs compared to traditional methods by providing a smoother gradient.

In the Improved WGAN, a gradient penalty replaces the original gradient clipping to enforce the 1-Lipschitz continuity more effectively. This modification addresses the instability and mode collapse issues observed in traditional GANs. The discriminator's loss function in Improved WGAN includes an additional term, representing this gradient penalty. The equation for the discriminator's loss function is given by

$$L(x^{(i)}, G_\theta(z^{(i)}); \phi) = \underbrace{-D_\phi(x^{(i)}) + D_\phi(G_\theta(z^{(i)}))}_{\text{original WGAN loss}} + \underbrace{\alpha \left(\left\| \nabla_{\hat{x}^{(i)}} D_\phi(\hat{x}^{(i)}) \right\|_2 - 1 \right)^2}_{\text{gradient penalty}}, \tag{3}$$

where α is a hyperparameter, and $\hat{x}^{(i)}$ is a linear combination of real and generated data points.

This improved model promises a more stable training process and generates high-quality synthetic instances of JSSPs, crucial for creating realistic and challenging scheduling problems.

2.4 Deterministic Policy Gradients

Creating realistic instances can be achieved with a basic GAN framework, where the generator is trained to transform a prior distribution into the data distribution. However, generating realistic JSSP instances is only half of the problem. We also aim to produce more challenging instances that could be used as novel benchmarks or to enhance the quality of a training dataset. To achieve this, we optimize the generation process using reinforcement learning to meet this non-differentiable metric.

In this context, a stochastic policy in reinforcement learning, represented as $\pi_\theta(s) = p_\theta(a|s)$, chooses actions based on a probability distribution. In contrast, a deterministic policy, represented as $\mu_\theta(s) = a$, selects actions directly. Following the MolGAN approach [15], we use a deterministic policy gradient algorithm. Specifically, an adapted version of deep deterministic policy gradient (DDPG), an off-policy actor-critic method, and which can be thought of as being deep Q-learning for continuous action spaces.

Unlike traditional models, this system does not evaluate state-action combinations, focusing solely on the graph G. We introduced a trainable, differentiable approximation of the reward function, $\hat{R}_\psi(G)$, to predict immediate rewards (i.e. the difficulty of the instance). The generator is then trained to maximize $\hat{R}_\psi(G)$, providing a gradient that guides the policy towards optimizing the desired metric. In practice, this can be viewed as utilizing a second discriminator that approximates a non-differentiable metric. However, one important difference is that the reward function and the generator's objectives are not adversarial. Thus, we can first train the reward network independently.

3 Methods

3.1 Difficulty Definition

The concept of the difficulty of a JSSP is inherently ambiguous. This ambiguity arises because an instance that poses considerable challenges for one algorithm may be effortlessly solved by another. To the best of our knowledge, there is no existing literature that explicitly discusses a methodology for quantifying the difficulty of an instance. Addressing this gap, our research proposes a novel approach to quantify difficulty.

We suggest measuring difficulty as the value of the best solution obtained by a standard solver within a time limit, normalized against a lower bound estimate of the makespan. This method allows for a comparative analysis of relative difficulty across varying instance sizes and temporal scales. The calculated ratio can be adjusted by subtracting one, ensuring the scale commences at zero.

$$\text{difficulty} = \frac{\text{best makespan found}}{\text{lower bound}} - 1 \tag{4}$$

This definition can be seen as a generalization of the normalized optimality gap (using the optimal makespan of the instance as the lower bound). If the solution found within the time limit is far from the optimal one, it makes sense to state to consider that problem as more difficult to solve. However, due to the inherently complexity of the JSSP, finding the optimal solution for a large enough number of instances can be too costly in practice. This is the reason we propose replacing the optimal makespan by another lower bound.

There are many approaches for obtaining good lower bounds. Some of them rely on relaxing some constraints of the problem [24]. In this work we opt for a simpler lower bound:

$$\text{lower bound} = \max \left(\max_{J_j \in J} \sum_{i \in J_j} d_{ij}, \max_{M_i \in M} \sum_{j \in M_i} d_{ij} \right) \quad (5)$$

To compute the lower bound we take the maximum of all job and machines total processing times. The total processing time of a job or machine is the sum of the duration of all operations associated with it. Each of these total processing times is a lower bound, because it represents what the makespan would be if there were no more operations other than the ones associated with that job or machine. We choose the best lower bound among these by taking the maximum value.

3.2 Dataset

Two datasets were created using two distinct methods, with some restrictions applied to the range of created instances for simplification. In each instance, there are exactly 10 machines. The number of jobs in each instance varies between 10 and 20. Additionally, every job is required to visit each machine exactly once.

Bounding the range of generated instances to these restrictions, two methods were used:

- **Purely Random Dataset.** A total of 49,519 instances were created using the naive random generated method. See algorithm 1 in the appendices section for more details.
- **Augmented Dataset**. This dataset has a total of 21,300 instances. It was created from a collection of well-known 162 benchmark instances by applying transformations to reduce their size and adding noise to each operation's duration. Further details, such as a more detailed explanation of the algorithm and the benchmark instances used for the generation of this dataset, are provided in Appendix 1.2.

After the creation of this two datasets, each instance is labeled using equation 4. Where the denominator is the lower-bound computed with equation 5, and the numerator is the best makespan found by a constraint programming (CP) solver with a time limit of 0.1 seconds running in a single 12th Gen Intel© Core™ i7-1265U processor. In particular, we used the CP-SAT solver from Google's ortools suite. This solver executes a multitude of diverse algorithms and strategies. This is a desirable feature since it makes our difficulty metric more strategy-agnostic.

We have chosen this time limit because optimal solutions are rarely found with these settings, yet it is sufficient time to find at least one solution for most instances (see Table I). Instances where no solution was found within the time limit have been assigned a difficulty of one, which represents the maximum difficulty value present in both datasets.

In order to have an accurate representation of the difficulty of each instance, it is important to have both the *optimal solution* and the *no solution* rate as low as possible. This is because the measure does not account for how quickly the solver reaches the optimal solution. Thus, comparing the difficulty of solved instances becomes less informative. The difficulty of the instance is determined by how close the optimum is to the lower bound, a concept similar to the scheduling efficiency of a solution. To assess the scheduling efficiency, we should use the average total processing times of all machines as the lower bound [25]. In this work, we take the maximum of all total processing times of jobs and machines since we are trying to estimate the optimality gap.

Dataset	Optimal Solutions	No Solution	Total
Purely Random	1719 (3.5%)	36 (0.1%)	49,519
Augmented	790 (3.7%)	0	21,300

TABLE I: Comparison of the solutions found by the constraint-programming-based solver with a time limit of 0.1 seconds

3.3 Model Architecture

Our architecture (Figure 2) follows the principle of MolGAN's framework, originally designed for molecular graph generation and adapted for generating JSSPs. The generator G_θ takes input from a prior distribution and generates a graph representing a job shop scheduling instance. The discriminator D_ϕ is trained to differentiate between instances from the training set and those generated by the generator. Both the generator and discriminator are trained using an improved WGAN approach, enabling the generator to match the empirical distribution of realistic and challenging scheduling problems.

The reward network $\hat{R}_\psi$ is a crucial addition, designed to optimize the generation of scheduling instances towards specific, non-differentiable metrics. In this case, their difficulty, which measures how challenging to solve the instance is. This network evaluates both real and generated instances and assigns a reward based on how well they meet the desired criteria.

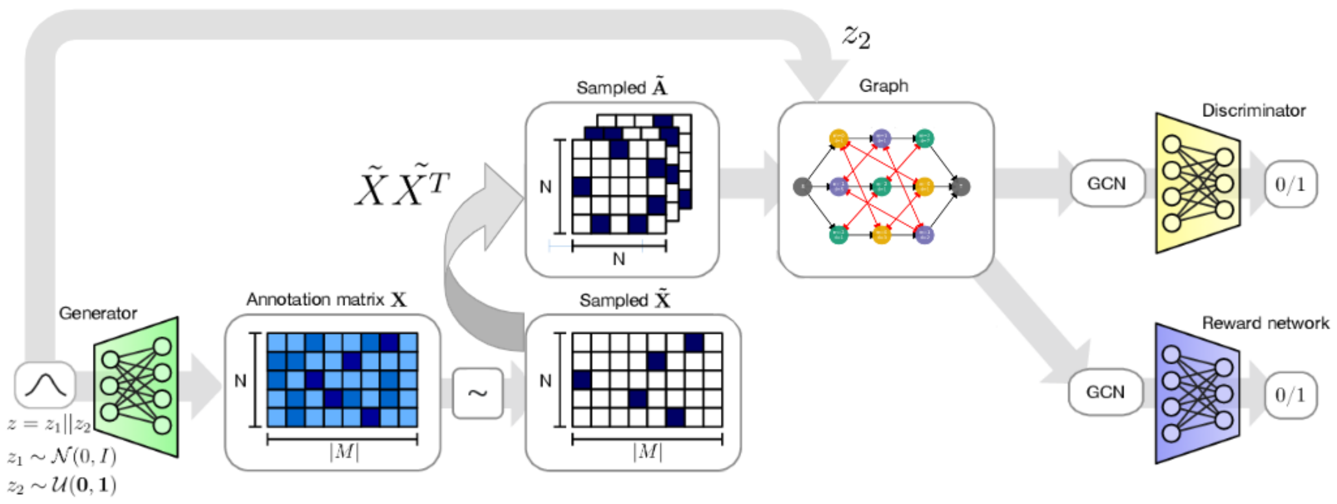
The discriminator is trained using the WGAN loss, while the generator uses a linear combination of the WGAN loss and the reinforcement learning (RL) loss. A hyperparameter λ in the range $[0, 1]$ regulates the trade-off between the WGAN and RL components.

$$L(\theta) = \lambda \cdot L_{WGAN}(\theta) + (1 - \lambda) \cdot L_{RL}(\theta), \quad (6)$$

This hybrid training approach ensures that the generated scheduling instances are not only realistic but also aligned with the desired properties of difficulty and complexity.

3.3.1 Generator

The generator G_θ takes a $(D + N)$-dimensional vector z and outputs graphs representing job shop scheduling instances. The vector z is composed of two vectors which are concatenated: a latent graph representation $z_1 \in \mathbb{R}^D$ sampled from a standard normal distribution ($z_1 \sim \mathcal{N}(0, I)$), and $z_2 \in \mathbb{R}^N$ sampled from a continuous uniform distribution ($z_2 \sim \mathcal{U}(\mathbf{0}, \mathbf{1})$), where N is the number of operations or nodes. The second vector z_2 represents the processing time or duration of each operation in the range $[0, 1]$.



z_2
Graph
Sampled $\tilde{\mathbf{A}}$
$\tilde{X}\tilde{X}^T$
N
N
GCN
Discriminator
0/1
GCN
Reward network
0/1
Generator
Annotation matrix $\mathbf{X}$
N
$|M|$
$\sim$
Sampled $\tilde{\mathbf{X}}$
N
$|M|$
$z = z_1||z_2$
$z_1 \sim \mathcal{N}(0, I)$
$z_2 \sim \mathcal{U}(\mathbf{0}, \mathbf{1})$

Fig. 2: Outline of our adjacency-matrix-based GAN architecture. From *letf*: the generator takes a sample z_1 from a prior distribution concatenated with some predefined information about the instance that must be generated (i.e. each operation's duration) z_2. This vector is processed by a MLP and an annotation matrix X with the probabilities of each operation belonging to each machine. It assigns a machine to each operation by sampling from the predicted categorical distribution. Then, the adjacency matrix $\tilde{A}$ of the disjunctive edges is created ($\tilde{A} = \tilde{X}\tilde{X}^T$), while the conjunctive edges' matrix is fixed in every generation. Both $\tilde{A}$ and each operation's duration z_2 represent a JSSP instance's disjunctive graph. Finally, the graph is processed by both the discriminator and reward networks, which are based on Relational-GCN layers [16]. This figure was created by modifying Figure 2 from N. D. Cao and T. Kipf's 2018 paper [15], which describes the MolGAN model architecture.

Note that this second vector is optional. Alternatively, the model could be designed to assign durations to each operation itself. However, this approach would add complexity, requiring the network to learn the distribution of processing times. Furthermore, adding the processing times as an additional random vector can be beneficial to add more variability to the generated instances. Letting the network predict this, could cause the network to always generate a similar set of durations for every instance.

In contrast to the MolGAN architecture [15], in which the model simultaneously outputs two continuous and dense objects: $X \in \mathbb{R}^{N\times T}$ for node types and $A \in \mathbb{R}^{N\times N\times Y}$ for edge types, where Y is the number of edge types. We separate this process in three steps:

1) The generator predicts the probability of each machine being associated to each operation, generating the matrix $X \in \mathbb{R}^{N\times T}$, where T is the maximum number of machines that each instance will have. A multi-layer perceptron (MLP) is used for simplicity and efficiency.
2) We sample from these categorical distribution using the Gumbel-Softmax trick [26] during the forward pass, but keeping the original continuous-valued tensors in the backward pass. This new matrix is denoted as $\tilde{X}$.
3) Finally we need to generate the adjacency tensor $\tilde{A} \in \mathbb{R}^{N\times N\times Y}$. Therefore, we need a different adjacency matrix for each type. The conjunctive adjacency matrix does not need to be learned since, the number of operations for each job is restricted to be the number of machines $|M|$, and, thus, the conjunctive edges always follow the same structure. On the other hand, the disjunctive adjacency matrix needs to be learned. However, since disjunctive edges connect operations that require the same machine, we can use the predicted machine for each node to generate this matrix following this rule. A simple way to compute the matrix is taking the product $\tilde{X}\tilde{X}^T$. This works because if two vectors have the same one-hot representation of the machine, their scalar product is one. In contrast, if they belong to different machines, since their respective vectors are orthonormal between them.

This changes to the architecture take advantage of the specific properties of JSSP's disjunctive graphs and allow us to output graphs an order of magnitude bigger. MolGAN's design was suited to output graphs of up to 20 nodes. We raised this number to 200 while reducing the number of output neurons in the MLP.

In order to generate problems with variable numbers of machines or jobs, the inputs of the MLP corresponding to non-existent operations can be masked. This is achieved by setting the last $N_{max} - N$ values of z_2 to zero. Additionally, we ignore the last $(N_{max} - N)|M|$ values of the MLP prediction. Here, N_{max} represents the maximum number of nodes our graph can have. We can compute the number of nodes of each JSSP's disjunctive graph multiplying the number of jobs $|J|$ with the number of machines $|M|$. This multiplication is based on the assumption that each job will pass through every machine exactly once, so it will have $|M|$ operations.

3.3.2 Discriminator and Reward Network

Both the discriminator D_ϕ and the reward network $\hat{R}_\psi$ take a graph as input and output a scalar value. They have similar architectures but do not share parameters. A series of graph convolution layers convolve node signals $\tilde{X}$ using the graph adjacency tensor $\tilde{A}$. The architecture is based on

Dataset	Count	Mean	Standard Deviation	Minimum	25%	50%	75%	Maximum
Benchmark	162	0.4848	0.3305	0.0000	0.2288	0.4484	0.6680	1.0000
Benchmark (10 machines, 10 to 20 jobs)	33	0.3720	0.1941	0.0649	0.1916	0.3472	0.5136	0.7755
Purely Random Dataset	49519	0.2033	0.0962	0.0000	0.1361	0.1975	0.2635	1.0000
Augmented Dataset	21300	0.2148	0.1072	0.0000	0.1426	0.2050	0.2727	0.8506

TABLE II: Comparison of Benchmark, Purely Random, and Augmented Datasets

the one proposed in the MolGAN paper [15]. We employ Relational-GCN layers, which supports multiple edge types in graphs.

Each node embedding at layer l is denoted as $h_i^{(l)}$, and the node embedding for the next layer is computed as:

$$\begin{aligned} \mathbf{h}_i'^{(l+1)} &= f_s^{(l)}(\mathbf{h}_i^{(l)}, \mathbf{x}_i) + \sum_{j=1}^{N}\sum_{y=1}^{Y} \frac{\tilde{A}_{ijy}}{|N_i|} f_y^{(l)}(\mathbf{h}_j^{(l)}, \mathbf{x_j}), \\ \mathbf{h}_i^{(l+1)} &= \tanh(\mathbf{h}_i'^{(l+1)}), \end{aligned} \tag{7}$$

where $\mathbf{h}_i^{(l)}$ represents the feature vector of node i at the l-th layer of the network. The function $f_s^{(l)}$ is a linear transformation that serves to create self-connections within layers, allowing each node to retain its own features during the transformation process. Additionally, $f_y^{(l)}$ is an edge type-specific affine transformation function at each layer.

The term N_i denotes the set of neighboring nodes for node i, and the normalization factor $1/|N_i|$ is used to scale the feature aggregation according to the number of neighbors. This normalization ensures that the resulting feature vectors are consistent in scale, preventing nodes with many neighbors from having disproportionately large influence due to the sum over a larger set of neighbors.

Finally, node embeddings are aggregated into a graph-level representation, which is further processed by an MLP to produce a scalar output for both the discriminator and the reward network. This aggregation is done following the work by Li et. al. (2016) [27]:

$$\begin{aligned} \mathbf{h}_G' &= \sum_{v\in V} \sigma\left(i\left(\mathbf{h}_v^{(L)}, \mathbf{x}_v\right)\right) \odot \tanh\left(j\left(\mathbf{h}_v^{(L)}, \mathbf{x}_v\right)\right), \\ \mathbf{h}_G &= \tanh\left(\mathbf{h}_G\right), \end{aligned} \tag{8}$$

where $\sigma(x) = 1/1+\exp(-x)$ denotes the logistic sigmoid function, and $\odot$ represents element-wise multiplication. The functions i and j are MLPs with linear outputs. They process concatenated inputs $\mathbf{h}_v^{(L)}$ and $\mathbf{x}_v$. In this context, $\sigma\left(i\left(\mathbf{h}_v^{(L)}, \mathbf{x}_v\right)\right)$ functions as a soft attention mechanism, highlighting node significance for the graph-level task.

4 Experiments and Results

4.1 Difficulty Distribution

In this section, we show the distribution of the difficulty found in each dataset. This is useful for evaluating if the computed difficulty score models correctly how challenging is to solve a particular instance. Furthermore, understanding the data distribution is crucial to understand the model results.

To assess the quality of the introduced difficulty score we can compare the difficulties computed for the benchmark dataset and the ones computed for our datasets. See Appendix 1.2 for more information about the instances composing this dataset.

4.1.1 Benchmark Dataset

In Figures 3 and 4 we show how the difficulty scores are distributed across the different benchmark instances. In order to compute their difficulty scores we followed the same procedure that was mentioned before. Figure 3 uses all benchmark instances. This dataset contains a very diverse set of problem sizes, ranging from instances with 6 machines and 6 jobs, to 20 machines and 100 jobs. Therefore, it is expected for smaller instances to be solved within the time limit while not finding a solution for the bigger ones. This is the reason we find a peak for the values zero and one. Note that instances where the difficulty score is zero are those which were solved within the time limit and their lower bound coincides with the optimum solution.

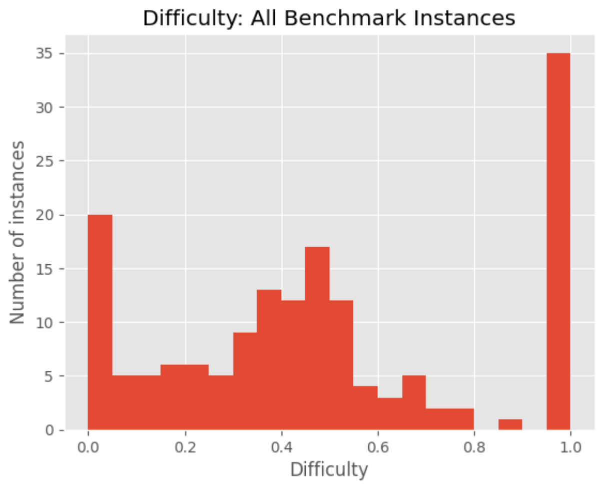


Fig. 3: Histogram of difficulty scores across all benchmark instances, computed using 20 bins.

The introduced difficulty score, however, has been adjusted primarily for instances of medium size (10 machines and 10 to 20 jobs). This is the reason it is also useful to filter the benchmark instances necessary to include only those which meet these restrictions. In this case, no instances with difficulties of zero or one were found.

4.1.2 Purely Random Dataset

In Figure 5 we can see that the distribution scores follow a Gaussian distribution centered around 0.2. As mentioned

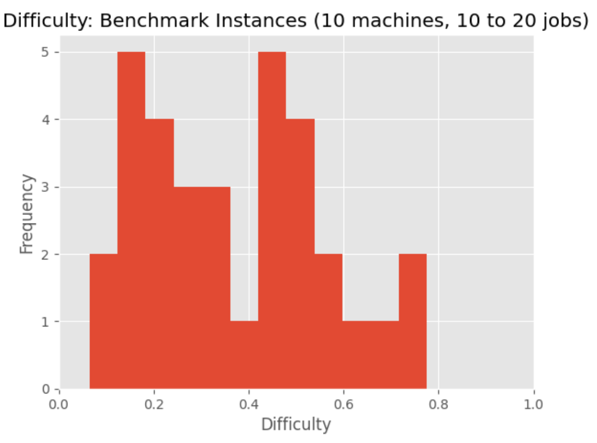


Fig. 4: Histogram of difficulty scores across all benchmark instances with 10 machines and between 10 and 20 jobs, computed using 12 bins.

in I, the solver could not find any feasible solution for 36 instances, this explains why there are some values of one.

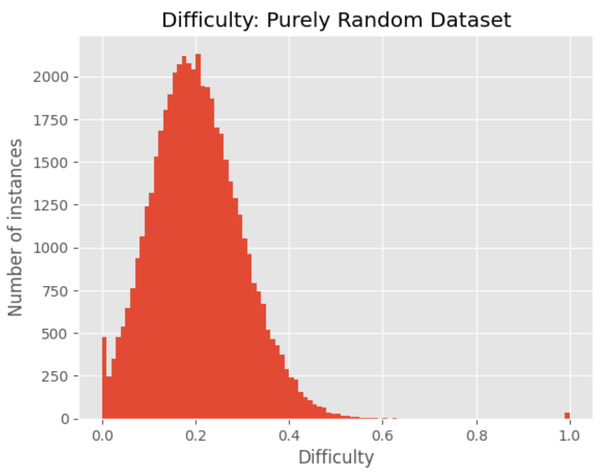


Fig. 5: Histogram of difficulty scores across all purely random generated instances, computed using 100 bins.

4.1.3 Augmented Dataset

This dataset was created by transforming and reducing the size of the aforementioned benchmark instances. In this case, the distribution seems similar to the Purely Random Dataset. This suggests that the perturbations made to these instances were enough to ”break” the characteristics that made these instances difficult to solve. We hypothesize that the reason why an instance is more challenging to solve than other could be very subtle. Sometimes a specific configuration of operation’s durations could be the main reason behind a high difficulty. Perturbing slightly them could make the instance easier to solve (and vice versa). Nevertheless, we can observe a larger ”tail”, and a slightly larger mean and median (see Table II). Since the main structure of the benchmark instances is preserved despite the transformations, this seems to indicate that their structure is slightly correlated with higher difficulty scores.

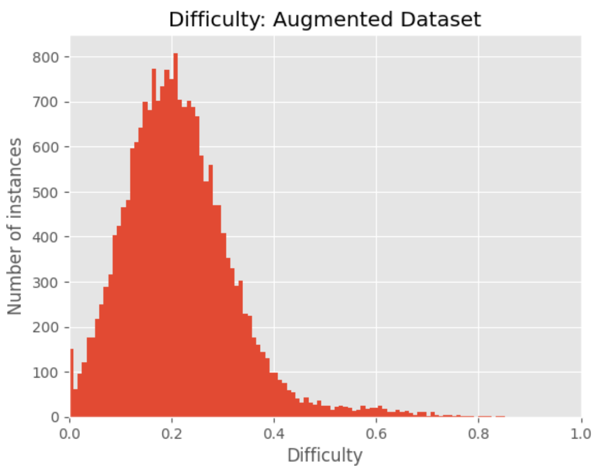


Fig. 6: Histogram of difficulty scores across all augmented instances, computed using 100 bins.

4.2 Difficulty Prediction

To evaluate the presented architecture, we first need to identify a GNN model that effectively models the complexity of a disjunctive graph. This involves selecting a suitable architecture and hyperparameters, achieved by training the proposed reward network to predict the difficulty of an instance. This section presents the results obtained in this regard.

We have found this task more challenging than expected. As it was hypothesized in section 1.2, we suspect that how challenging it is to solve a JSSP could depend on subtle and complex combinations of durations and distribution of machines. Thus, the task of measuring this becomes extremely difficult. These challenges have made impossible to execute a complete set of experiments that implement the proposed architecture before the deadline of this project. The proposed reward network architecture, despite its features, seems to do not be expressive enough to capture the features of this problem. In this section, we share some of the approaches followed and their results.

We found this task more challenging than anticipated. As hypothesized in Section 1.2, the difficulty in solving a JSSP may depend on subtle and complex combinations of durations and machine distributions. Consequently, accurately measuring this is extremely difficult. These challenges prevented us from implementing the proposed architecture before this project’s deadline. This section shares some of the approaches we followed and their results.

4.2.1 Pooling-based Graph Aggregation Layer and Leaky ReLU

Selecting and designing optimal aggregation operations is an open research topic [28]. In our initial experiments, we used the following graph aggregation layer:

$$\mathbf{h}_G = [GMP(G) || \sum_{v \in V} \mathbf{h}_v || \frac{1}{|G|} \sum_{v \in V} \mathbf{h}_v] \tag{9}$$

where $GMP(G)$ denotes a Global Max Pooling operation and $||$ represents concatenation. This aggregation layer combines different types of aggregations to enhance expressiveness. However, it's important to note that if $\mathbf{h}_G$'s values are predominantly positive, their sum could yield very large numbers. To mitigate this, we employed a Leaky ReLU activation function with α set to 0.1, as commonly used in GANs for images. Unlike ReLU, Leaky ReLU multiplies negative values by α. Nonetheless, empirical results indicated a significant issue of vanishing gradients when using a sigmoid function as the output layer. Figure 7 illustrates this, showing nearly zero gradient distribution at almost every step for the first R-GCN layer, although similar observations could be made for other layers.

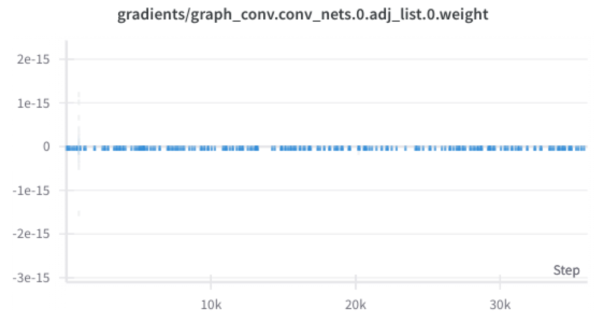


Fig. 7: Gradient distribution in the first R-GCN layer, as viewed on the Weights and Biases Dashboard. The batch size used is one, with each step representing a weight update.

The vanishing gradient problem arises because the network produces extreme values for each instance before the sigmoid activation function. Since the gradient in this region is almost zero, the network fails to update its parameters adequately. Partially resolving this problem involves using a linear activation instead of a sigmoid for the output layer. Figure 8 supports this hypothesis, showing the network outputting values exceeding one thousand during early iterations.

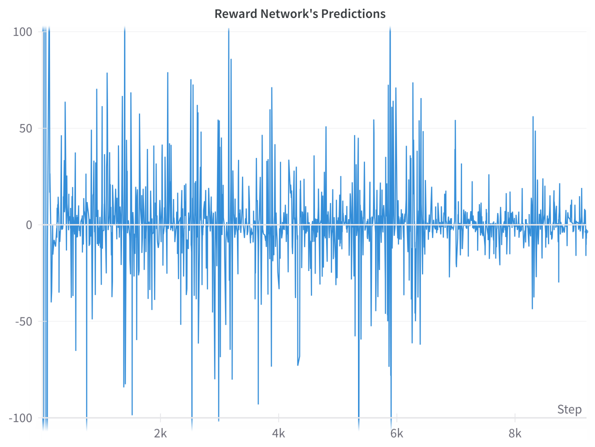


Fig. 8: Difficulty score predictions by the reward network during training with the sigmoid activation removed from the output layer. Screenshot from the Weights and Biases Dashboard. The batch size used is one.

Although replacing the sigmoid with a linear activation function in the last layer addressed the gradient vanishing issue, it did not resolve the underlying instability of the network. We suspect that Leaky ReLU, despite allowing for negative values, fails to prevent passing excessively large numbers to the final MLP following the aggregation layer. This led us to use the aggregation layer described in Equation 3.3.2.

4.2.2 Soft-Attention-Based Aggregation Layer

Employing the aggregation layer from Equation 3.3.2, as explained in the previous section, significantly enhanced the network's stability. We also replaced the Leaky ReLU activation function with the hyperbolic tangent (tanh). Tanh, which outputs values between -1 and 1, is an excellent alternative to Leaky ReLU for preventing the input of overly large numbers to the final MLP.

However, even after these improvements, we were unable to extract sufficiently meaningful results. As can be seen in Figure 9, the network's predictions remained relatively constant, hovering around the mean difficulty value of the dataset's instances.

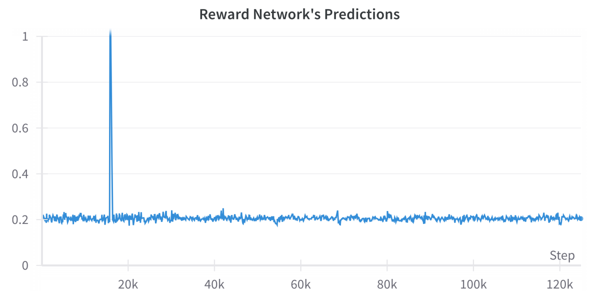


Fig. 9: Difficulty scores predictions by the reward network during training with the soft-attention-based aggregation layer. Screenshot taken from the Weights and Biases Dashboard. The batch size used is one.

Some additional details about the tested hyperparameters include:

- **Changes in the Initial Node Features**: The initial node features that encoded each operation were as follows: its duration normalized within the range [0, 1], and its job represented by its normalized index (we opted not to use a one-hot representation due to its variable size). Using the one-hot embedding of the machine to which it belonged did not yield beneficial results. However, additional features, such as node2vec [29], were not tested.
- **Changes in Model Hyperparameters**: This was the most extensively explored option. We conducted a total of 56 runs, logged with the Weights and Biases platform. The initial experiments utilized three R-GCN layers with sizes [16, 32, 64], respectively. Other configurations tested included [6, 12], [64, 32], and [128, 128, 128]. Regarding the graph layer aggregation, we experimented with sizes ranging from 24 to 256. Lastly, we also explored various structures for the final MLP, testing configurations such as [128, 68, 32, 1], [128, 64, 1], [24, 12, 1], and [128, 1].

- **Learning Rate**: We consistently used the Adam optimizer, experimenting with learning rates of 0.0001 and 0.001.

5 Future Work

The research presented in this paper lays the foundation for numerous directions in the generation and analysis of JSSPs. Future work should focus on enhancing the current framework and validating its effectiveness through comprehensive experiments and comparisons with existing models.

The challenges encountered in accurately predicting the difficulty of JSSP instances with our current GNN approach suggest that there is considerable scope for architectural improvements. A promising direction could be the incorporation of Graph Transformer models [30], known for their effectiveness in capturing long-range dependencies and complex patterns in data. Furthermore, future research could focus on identifying and extracting more informative node features that can better represent the characteristics and complexities of JSSP instances. This could involve exploring advanced feature engineering techniques, incorporating domain knowledge, and leveraging unsupervised learning methods like node2vec [29] to enrich the representation of nodes in the graph. Other unsupervised learning methods like pre-training in a different task could also help to solve the current issues.

Once improvements in the architecture and feature representation are achieved, it will be crucial to conduct thorough empirical experiments to validate the effectiveness of the GAN framework. These experiments should aim to demonstrate that the framework can generate realistic and challenging JSSP instances, and that it offers advantages over traditional instance generation methods. Trying to contribute in this regard, we created an augmented dataset from a series of benchmark instances. This dataset does not preserve the difficulty score distribution of the original benchmark, however their instance follow similar structures by design. Therefore, this dataset could be used as true labels for assessing the realism of instances within the GAN framework. However, a contribution with real world instances would be very valuable in this regard.

6 Conclusions

This paper has introduced a novel framework for generating realistic and challenging Job Shop Scheduling Problems (JSSPs) using an adjacency-matrix-based Generative Adversarial Network (GAN) adapted from a molecular graph generation model.

We proposed a new way to measure the difficulty of JSSP instances. This metric is based on the best solution obtained by a standard solver within a time limit, normalized against a lower bound estimate of the makespan. This measure provides a relative scale of difficulty, useful for comparing instances of varying sizes and complexities. Benchmark instances, selected for its difficulty, were successfully classified as more difficult to solve than naively generated, supporting the usefulness of the metric. Additionally, two datasets containing a total of 70,819 instances were generated and labeled with difficulty scores.

Our experiments revealed the challenges involved in predicting the difficulty of JSSP instances using GNNs. Despite employing various architectures and hyperparameters, the task proved more complex than anticipated. Thus, the findings open up new avenues for future research, particularly in exploring more advanced GNN architectures and learning strategies to better understand and predict the complexity of JSSP instances.

Appendix I
Dataset

1.1 Puerely Random Dataset

The naive algorithm for generating JSSP instances operates as follows:

Algorithm 1 Naive Algorithm for Generating JSSP Instances

```
Output: JSSP
 1: $N_{jobs} \sim \mathcal{U}(10, 20)$
 2: $N_{machines} \leftarrow 10$
 3: jobs $\leftarrow \{\}$
 4: for $j = 1$ to $N_{jobs}$ do
 5:     job $\leftarrow \{\}$
 6:     available_machines $\leftarrow \{1, \ldots, N_{machines}\}$
 7:     for $m = 1$ to $N_{machines}$ do
 8:         machine_id $\sim \mathcal{U}$(set of available machines)
 9:         duration $\sim \mathcal{U}(1, 100)$
10:         operation $\leftarrow$ (machine_id, duration)
11:         job $\leftarrow$ job $\cup$ {operation}
12:         Remove machine_id from available_machines
13:     end for
14:     jobs $\leftarrow$ jobs $\cup$ {job}
15: end for
16: return new JSSP(jobs)
```

It first determines the number of jobs randomly within the range of 10 to 20. Then, for each job, it randomly assigns a machine (from a total of 10 machines) and a duration (between 1 to 100) to each operation, ensuring each machine is used exactly once per job. After defining all operations for each job, it compiles these jobs to create a new JSSP instance.

1.2 Augmented Dataset

The instances used to perform the data augmentation come from a collection of benchmark datasets. The contributions to this benchmark dataset are as follows:

- **abz5-9**: This subset, comprising five instances, was introduced by [31].
- **ft06, ft10, ft20**: These three instances are attributed to the work of Fisher and Thompson, as detailed in [32].
- **la01-40**: A collection of forty instances, this group was contributed by Lawrence, as referenced in [33].
- **orb01-10**: Ten instances in this category were provided by Applegate and Cook, as seen in [34].
- **swb01-20**: This segment, encompassing twenty instances, was contributed by Storer et al., as per [35].
- **yn1-4**: Yamada and Nakano are credited with the addition of four instances in this group, as found in [36].
- **ta01-80**: The largest contribution, consisting of eighty instances, was made by Taillard, as documented in [12].

Additionally, the metadata of these instances has been updated using data from [37].

The transformations applied to each benchmark instance was the following:

1) **Reduction to 10 Machines:** For each original instance, the number of machines is reduced to 10. If an instance has more than 10 machines, new instances are generated by removing machines until the desired number is reached. If it already has 10 machines, it is maintained as is.
2) **Job Adjustment:** Then, for each resulting instance from the first step, the number of jobs is adjusted to be between 10 and 20. If an instance has more than 10 jobs, new instances are generated by removing jobs until a number within the desired range is reached.
3) **Adding Noise to Durations:** Finally, noise is added to the job durations of each resulting instance from the second step to increase data diversity.

Algorithm 2 Algorithm for Augmenting a JSSP Instance

```
Input: A single JobShopInstance object $i$
Output: augmented JSSP objects $\{i'_1, i'_2, \ldots, i'_n\}$
    Step 1: Reduction to 10 Machines
 1: $I_1 \leftarrow \{\}$
 2: if $i.n_{machines} > 10$ then
 3:     $i' \leftarrow$ removeMachines$(i, 10)$
 4:     $I_1 \leftarrow I_1 \cup \{i'\}$
 5: else
 6:     $I_1 \leftarrow I_1 \cup \{i\}$
 7: end if
    Step 2: Job Number Adjustment
 8: $I_2 \leftarrow \{\}$
 9: for each instance $i$ in $I_1$ do
10:     if $i.n_{jobs} > 10$ then
11:         for $k = 1$ to $i.n_{jobs} - 10$ do
12:             $i' \leftarrow$ removeJobs$(i, [10, 20])$
13:             $I_2 \leftarrow I_2 \cup \{i'\}$
14:         end for
15:     else
16:         $I_2 \leftarrow I_2 \cup \{i\}$
17:     end if
18: end for
    Step 3: Adding Noise to Durations
19: $I_3 \leftarrow \{\}$
20: for each instance $i$ in $I_2$ do
21:     $i' \leftarrow$ addNoiseToDurations$(i)$
22:     $I_3 \leftarrow I_3 \cup \{i'\}$
23: end for
24: return $I_3$
```

The function $removeMachines$ and $removeJobs$ remove machines or jobs, respectively, and all its associated operations randomly. The first ensures that the new instance has exactly 10 machines, while the latest that it has between 10 and 20 jobs. The function $addNoiseToDurations$ changes the duration of each operation by adding random noise sampled from $\mathcal{U}(-10, 10)$. The resulting values are then clipped to keep them in the $[1, 100]$ range.

Thus, the number of additional instances generated for each benchmark example is given by the expression:

$$\max\{(m-10)\times(n-10), n-10, 0\}$$

where m and n are the number of machines and jobs of the original instance, respectively.

1.3 Encoding for the CP-SAT Solver

We encode a JSSP as a linear programming problem in the following way:
Variables:
$start_{ij}$ and end_{ij} denote the start and end times of operation O_{ij}.
Objective:

$$\text{Minimize } makespan = \max_{i,j}(end_{ij}) \tag{10}$$

Constraints:
Job Sequencing Constraints:

$$end_{ij} \leq start_{i(j+1)}, \quad \forall i \in J, \forall j < n_i \tag{11}$$

Machine Non-Overlap Constraints:
For each $M_k \in M$, no two operations $O_{ij}$and O_{pq} assigned to M_k can overlap in time.

References

[1] H. Xiong, S. Shi, D. Ren, and J. Hu, "A survey of job shop scheduling problem: The types and models," *Computers Operations Research*, vol. 142, p. 105731, 2022.

[2] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Mathematics of Operations Research*, vol. 1, no. 2, pp. 117–129, 1976.

[3] B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*. Springer, 5th ed., 2012.

[4] I. Boussaïd, J. Lepagnot, and P. Siarry, "A survey on optimization metaheuristics," *Information Sciences*, vol. 237, pp. 82–117, 2013.

[5] V. V. Vazirani, *Approximation Algorithms*. Springer, 2010.

[6] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: A methodological tour d'horizon," *European Journal of Operational Research*, vol. 290, no. 2, pp. 405–421, 2021.

[7] J. Kotary, F. Fioretto, P. V. Hentenryck, and B. Wilder, "End-to-end constrained optimization learning: A survey," *CoRR*, vol. abs/2103.16378, 2021.

[8] Q. Cappart, D. Chételat, E. B. Khalil, A. Lodi, C. Morris, and P. Veličković, "Combinatorial optimization and reasoning with graph neural networks," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21* (Z.-H. Zhou, ed.), pp. 4348–4355, International Joint Conferences on Artificial Intelligence Organization, 8 2021. Survey Track.

[9] J. Błazewicz, E. Pesch, and M. Sterna, "The disjunctive graph machine representation of the job shop scheduling problem," *European Journal of Operational Research*, vol. 127, no. 2, pp. 317–331, 2000.

[10] C. Zhang, W. Song, Z. Cao, J. Zhang, P. S. Tan, and C. Xu, "Learning to dispatch for job shop scheduling via deep reinforcement learning," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, (Red Hook, NY, USA), Curran Associates Inc., 2020.

[11] W. Song, X. Chen, Q. Li, and Z. Cao, "Flexible job-shop scheduling via graph neural network and deep reinforcement learning," *IEEE Transactions on Industrial Informatics*, vol. 19, no. 2, pp. 1600–1610, 2023.

[12] E. Taillard, "Benchmarks for basic scheduling problems," *European Journal of Operational Research*, vol. 64, no. 2, pp. 278–285, 1993.

[13] G. D. Col and E. C. Teppan, "Large-scale benchmarks for the job shop scheduling problem," *ArXiv*, vol. abs/2102.08778, 2021.

[14] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014.

[15] N. D. Cao and T. Kipf, "Molgan: An implicit generative model for small molecular graphs," *ArXiv*, vol. abs/1805.11973, 2018.

[16] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," *arXiv preprint arXiv:1703.06103*, 2017.

[17] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[18] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *2nd International Conference on Learning Representations, ICLR 2014, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), (Banff, AB, Canada), 2014.

[19] D. J. Rezende, S. Mohamed, and D. Wierstra, "Stochastic backpropagation and approximate inference in deep generative models," in *Proceedings of the 31st International Conference on Machine Learning* (E. P. Xing and T. Jebara, eds.), vol. 32 of *Proceedings of Machine Learning Research*, (Bejing, China), pp. 1278–1286, PMLR, 22–24 Jun 2014.

[20] M. Simonovsky and N. Komodakis, "Graphvae: Towards generation of small graphs using variational autoencoders," *ArXiv preprint*, vol. abs/1802.03480, 2018.

[21] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, (Red Hook, NY, USA), p. 2234–2242, Curran Associates Inc., 2016.

[22] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved training of wasserstein gans," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 5769–5779, Curran Associates Inc., 2017.

[23] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *Proceedings of the 34th International Conference on Machine Learning* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 214–223, PMLR, 2017.

[24] P. Brucker and B. Jurisch, "A new lower bound for the job-shop scheduling problem," *European Journal of Operational Research*, vol. 64, no. 2, pp. 156–167, 1993.

[25] S. Mirshekarian and D. N. Šormaz, "Correlation of job-shop scheduling problem features with scheduling efficiency," *Expert Systems with Applications*, vol. 62, pp. 131–147, 2016.

[26] E. Jang, S. S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," *ArXiv*, vol. abs/1611.01144, 2016.

[27] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *4th International Conference on Learning Representations, ICLR 2016, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), (San Juan, Puerto Rico), 2016. Accessed: [Insert access date here].

[28] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," *ArXiv*, vol. abs/1810.00826, 2018.

[29] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), p. 855–864, Association for Computing Machinery, 2016.

[30] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim, "Graph transformer networks," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

[31] J. Adams, E. Balas, and D. Zawack, "The shifting bottleneck procedure for job shop scheduling," *Management Science*, vol. 34, no. 3, pp. 391–401, 1988.

[32] J. Muth and G. Thompson, *Industrial scheduling*. Englewood Cliffs, NJ: Prentice-Hall, 1963.

[33] S. Lawrence, "Resource constrained project scheduling: An experimental investigation of heuristic scheduling techniques (supplement)," tech. rep., Carnegie-Mellon University, Graduate School of Industrial Administration, Pittsburgh, Pennsylvania, 1984.

[34] D. Applegate and W. Cook, "A computational study of job-shop scheduling," *ORSA Journal on Computer*, vol. 3, no. 2, pp. 149–156, 1991.

[35] R. Storer, S. Wu, and R. Vaccari, "New search spaces for sequencing problems with applications to job-shop scheduling," *Management Science*, vol. 38, no. 10, pp. 1495–1509, 1992.

[36] T. Yamada and R. Nakano, "A genetic algorithm applicable to large-scale job-shop problems," in *Proceedings of the Second International Workshop on Parallel Problem Solving from Nature (PPSN'2)*, (Brussels, Belgium), pp. 281–290, 1992.

[37] T. Weise, "jsspinstancesandresults." https://github.com/thomasWeise/jsspInstancesAndResults, Accessed in January 2024.