650.060 – Small Project in AI Winter Term, 2023/24

A comparison between constructive and repairing approach to solve JSSP

Massimo Calabrigo 12247382

Contents

1 Introduction

Solving NP-Hard combinatorial problem has always been a task with lots of practical applications (Job Shop Scheduling in industrial settings, Vehicle Routing Problem for logistics), and traditionally these problems were solved either by looking for euristics, which are polynomial time algorithms that produces optimal results on typical inputs [6], or Costraint Optimization Methods (COM) such as Genetic Algorithms [3], Constraint Programming and Linear programming [4].

More recently another set of techniques based on Reinforcement Learning (RL) and Deep learning (DL) began to emerge (RL+DL), which is able to outperform [5] or be very near [1] the current state-of-art approaches. The most promosing advantage offered by RL+DL is the possibility of life-long learning [1], which enables the agent trained on a subset of instances of a problem, to also generalize the knowledge it learned to other instances aswell. Another relevant advantage of RL+DL, over aformentioned Constraint Optimization Methods, is that RL+DL can model the stochasticity of a problem, a property that can be easily seen in policy-gradient based RL architectures, in which, from a given state, the probability distribution of the actions to take is computed [1].

In this work we will look at two approaches used to solve NP-Hard problems with RL+DL: the constructive RL approach [1], which builds the solution of the problem from scratch, and the repairing approach [5], which iteratively improves a given solution, with our goal being to apply the repairing approach to the Job Shop Scheduling Problem (JSSP).

In section 2 we provide a description of JSSP and in section 3 we explain the constructive approach that was used in "A Reinforcement Learning Environment For Job-Shop Scheduling" [1] to solve JSSP, in section 4 we explain the repairing approach used in NeuRewriter [5], that was used to solve three NP-Hard problems: Halide Expression semplification (HES), Multi-Resource tasks (MRT) and Vehicle Routing Problem (VRP), focusing on the implementation of NeuRewriter to MRT, a similar problem to JSSP and in section 5, we make a comparison between the results obtained by the constructive approach on JSSP and the results obtained by NeuRewriter on the problems it solved, MRT especially.

Finally, in section 6, we enumerate a few possible implementations for a repairing-based RL+DL method to solve JSSP, and discuss in more detail one of them, also providing research questions to be investigated in future work.

2 Job scheduling

2.1 Job shop scheduling - JSSP

We are given m machines and n jobs, where each job J_i has at most as many operations as the number of machines: $o_{i,1}, \ldots, o_{i,k}$, $k \leq n$. Each operation $o_{i,j}$, is a tuple of the

form $(d_{i,j}, m_{i,j})$, where $d_{i,j} \in [1, T_{max}]$ is an integer value representing the duration of the operation, while $m_{i,j} \in [1, m]$ is an integer indicating the machine on which the operation $o_{i,j}$ has to be executed.

The set of operations of each job is ordered, and the operations of each job must be executed in this order. In addition, each machine is only able to execute at most one operation at a time, precluding the possibility of concurrency within the same machine, and the execution of an operation can be neither stopped nor resumed.

The objective is to find a schedule which minimizes either the makespan or the tardiness or the maximum lateness etc. [13]. We set the makespan as the objective function to minimize, which is the earliest time in which all jobs have been scheduled and executed.

Given the following JSSP instance

$$J_0 = \{(3,0),(2,1),(2,2)\}, J_1 = \{(2,0),(1,2),(4,1)\}, J_2 = \{(4,1),(3,2)\}$$

we provide two different solutions in Figure 1, along with their makespan. As we can see, operations belonging to the same job are executed in order and on different machines, with no parallelism being present on any single machine.

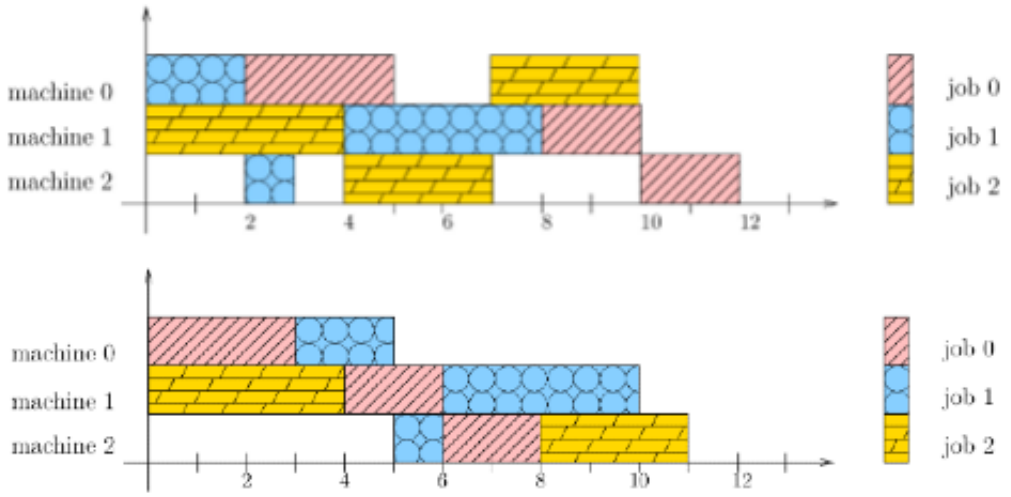

Figure 1: This example shows 2 different solutions of the aformentioned JSSP instance with 3 machines and 3 jobs [2]. In the bottom solution, the earliest time in which every operation has been executed is 11, hence the makespan of the solution is 11, while the top solution has a makespan of 12. We conclude that the bottom solution is better than the top one because it does a better job in minimizing the makespan.

3 Constructive approach to solving JSS

In this section we examine the constructive approach to solve JSSP, by analyzing the paper: "A Reinforcement Learning Environment For Job-Shop Scheduling"[1], which

adopts a constructive approach to solve the problem. Starting from no operation allocated, the constructive approach allocates one or more operations at each timestep until a solution is reached, and for this reason it is called "constructive".

3.1 Search space

In this section we explore the discrete search space for the JSSP, describing both the state encodings and actions, the reward function and the minimization objective of JSSP. We further explore the reason why the action NO-OP was added, and the implementation of search rules to aid the algorithm to correctly learn to use NO-OP.

3.1.1 States

Each state represents a scheduling at time t of an instance of JSSP, and it is encoded as a $|J| \times 7$ matrix M_1. The encoding contains 7 attributes for each job[1].

1. A Boolean to represent if the job can be allocated.
2. The left-over time for the currently performed operation on the job. This value is scaled by the longest operation in the schedule to be in the range $[0, 1]$.
3. The percentage of operations finished for a job.
4. The left-over time until total completion of the job, scaled by the longest job total completion time to be in the range $[0, 1]$.
5. The required time until the machine needed to perform the next job's operation is free, scaled by the longest duration of an operation to be in the range $[0, 1]$.
6. The IDLE time since last job's performed operation, scaled by the sum of durations of all operations to be in the range $[0, 1)$.
7. The cumulative job's IDLE time in the schedule, scaled as 6) to be in the range $[0, 1)$.

We provide as example the encoding of a solution to the following JSSP instance, $J_1 = \{(4, 1), (3, 2), (4, 0)\}$, $J_2 = \{(2, 0), (1, 2), (4, 1)\}$, $J_3 = \{(2, 0), (3, 1), (1, 2)\}$, at timestep $t = 7$, in Figure 2.

The matrix M_1, obtained from encoding the state s_1, can be decoded to a set of states $s_1, \ldots, s_k$, however, M_1 holds enought information to garantee that starting from any of the reconstructed states, equivalent solutions will be reached [1].
Each state s_i, is a couple defined by both a matrix M_i as described above, and an array of booleans B= $[b_1, \ldots, b_{|J|+1}]$ of size $|J| + 1$, which indicates which actions are legal and can be taken: $s_i = (M_i, B_i)$.

The concept of "legality of an action" is motivated by the need to better learn the NO-OP action, and it is explained in the next subsection.

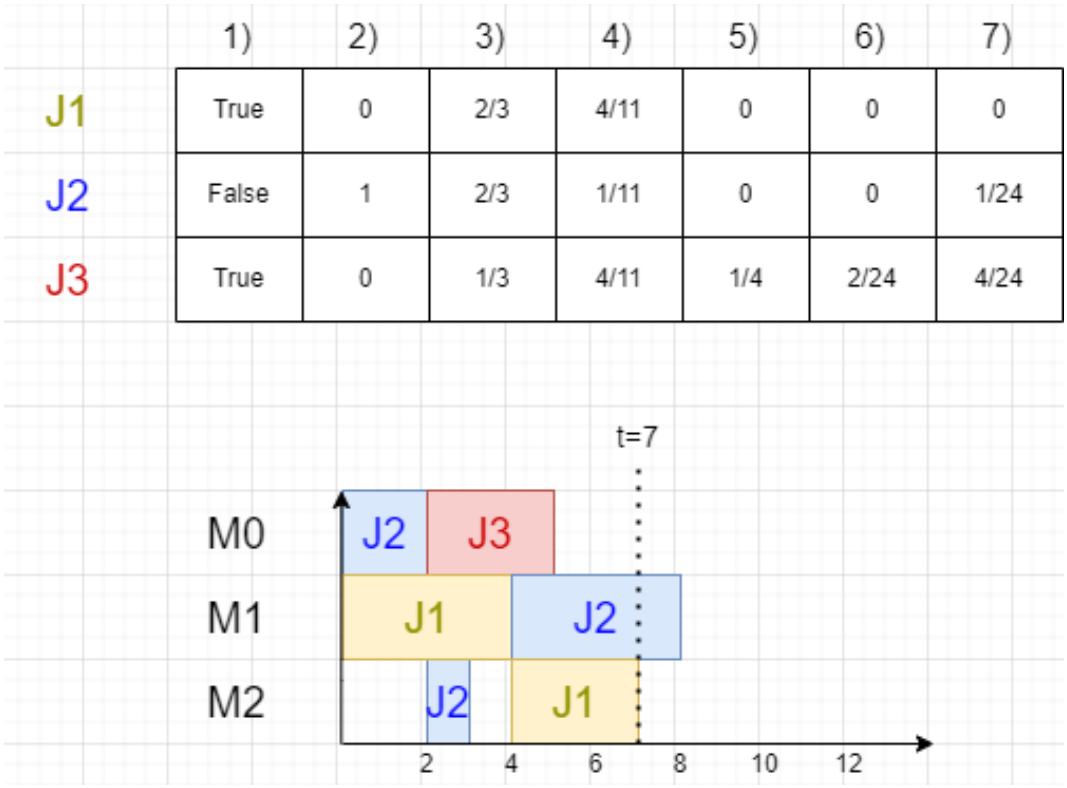

Figure 2: On the bottom a schedule at timestep $t = 7$ for the aformentioned JSSP instance is shown, and on the top, the $|J| \times 7$ matrix encoding for the same schedule is shown.

3.1.2 Actions

The set of actions for a state s_t contains $|J|$ operations $o_{i,j}$ (the current allocatable operation from each job J_i), plus an additional NO-OP action, which allocates no jobs: A= $\{J_1, \ldots, J_n$,NO-OP$\}, |A| = |J| + 1$. Adding the action NO-OP to the action set enables a wider search space because the algorithm is able to choose not to allocate a job, even if that job could be allocated, also enabling more complex strategies involving waiting in order to schedule a more suitable job, but the problem with NO-OP is that it is complex to learn, and if not learned correctly, could resulting in an algorithm with worst result than greedy policies [1].

In order to correctly learn NO-OP we make use of the boolean vector B of legal actions which is present in every state, and that effectively reduces the set of possible actions that can be taken from state s= (M, B), by implementing the following rules.

- **NO-OP Symmetry rule**: The symmetry introduced by NO-OP regarding the order of execution betweem the NO-OP action and an allocatable Job J_i in a given timeframe is broken, by imposing that, if at time t, operation NO-OP is taken when J_i was allocatable, then J_i becomes an illegal action ($B[i]$ =False) until another action is executed.

- **NO-OP suppression rule**: If there are 4 free machines, or the 15% or more of the jobs are allocatable, then we make the usage of NO-OP illegal until the conditions of the suppresion rule are no longer valid ($\forall i, B[i] = False$).

Implementing NO-OP gives not only the possibility of enabling more complex scheduling strategies, but also addresses the effect of symmetry caused by different schedules which leads to the same solution, but have different ordering of jobs.
It does so by forcing some ordering on the jobs allocation, due to NO-OP usage being dependent of the number of available machines and allocatable jobs [1].

3.1.3 Reward Function

The natural reward function for the JSSP task would be the makespan function, but due to the fact that the search space includes not only solutions, but also schedulings where not all jobs have been scheduled, using the makespan as reward function would result in a **sparse reward function**, given that it can give a reward only when all jobs have been scheduled.
To make up for this problem, the authors of "A Reinforcement Learning Environment For Job-Shop Scheduling" decided to use a dense reward function

$$r(s, a) = p_{a,j} - empty_m(s, s^{'})$$

which can be computed on every schedule in the search space, and they also analyze the relashionship between the goal of minimizing the makespan and maximizing their dense function, to make sure that the same goal is being achieved (Figure 3).

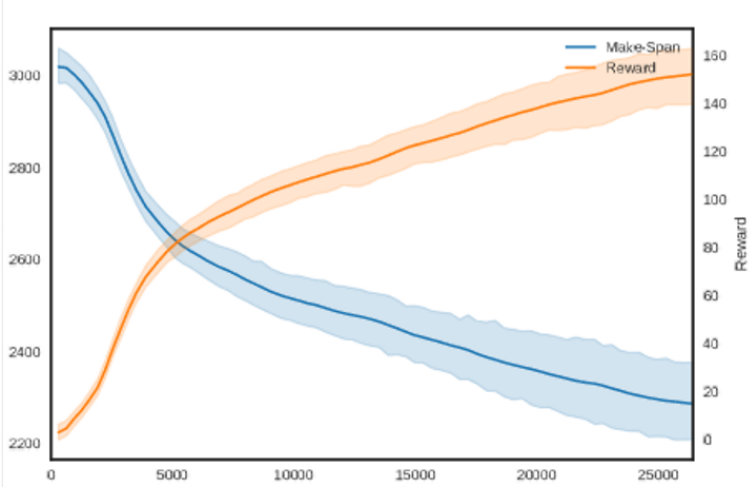

Figure 3: We can see there is a clear relashionship between the goal of minimizing the makespan (blue), and maximizing the dense reward function (orange) designed above [1]

3.2 Objective function and Architecture

The model chosen to solve JSSP instances with the constructive approach paper is PPO (Proximal Policy Optimization), combined with the Actor Critic Approach, see Figure 4.

Actor Critic methods are a combination of policy-based (actor) and value-based methods (critic), allowing for faster training than policy-based methods such as REINFORCE [8], but not being as unstable as value-based methods Q-Network [8].

A problem of Vanilla Actor Critic methods is that they tend to suffer from **policy-crash** if the step-size is inadeguate [1], so PPO is used as policy-based method instead of REINFORCE. PPO tries to take the biggest possible step without causing policy-crash [1], hence combining the advantages of faster training provided by using a Critic, and having a PPO actor that decreases the probability to cause policy-crash.

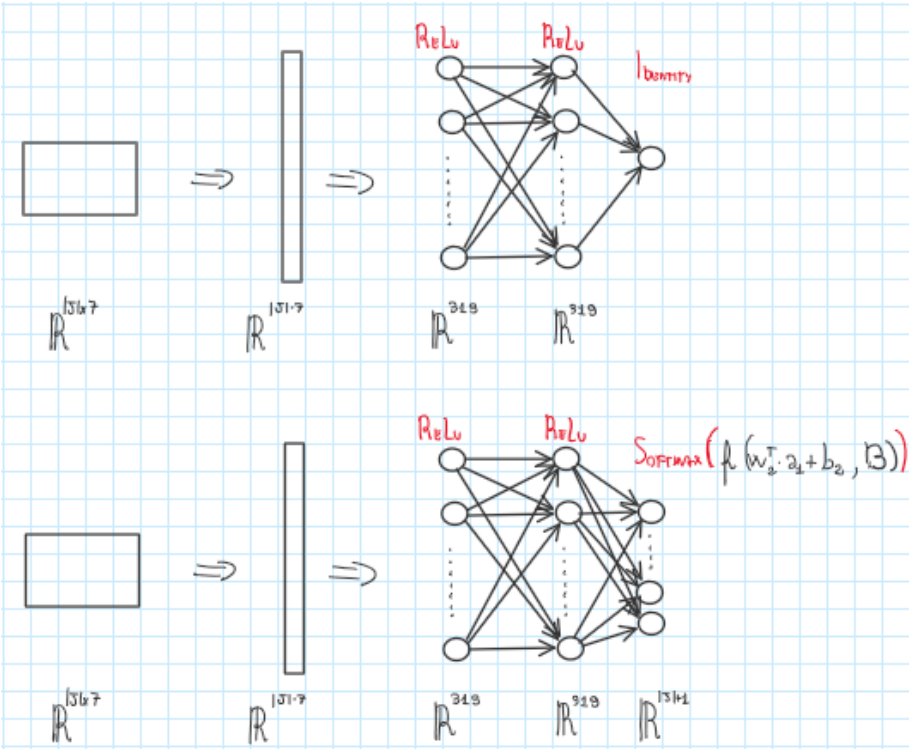

Figure 4: The state-value function on top, and the action selection network on the bottom, both takes as input the matrix M of the state $s = (M, B)$, while the action selection network also makes use of the vector of legal actions B.
We can see that in the action-selection network a function $f(network\_output, B)$ is used to reduce the network outputs, where B[i] is set to "illegal action", to small negative numbers before softmax is applied, doing so, illegal actions will have a close to zero probability of being allocated.

The objective function for the Actor Critic method with Clipped PPO policy-based method is expressed as follow:

$$L = min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), g(\epsilon, A^{\pi_{\theta_k}}(s,a))\right)$$

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & \text{if } A \geq 0 \\ (1-\epsilon)A, & \text{otherwise} \end{cases}$$

where θ and θ_k are the parameters of the new and old policies respectively [1].

4 Repairing approach to solving NP_Hard Problems

In this section we will examine the second RL+DL based method to solve NP-Hard problems: the repairing approach [5].

The repairing approach, or NeuRewriter, works on a search space constituted only by solutions of an instance of a problem, and instead of constructing a solution from scratch (like the constructive approach to solve JSSP 3), it learns an algorithm to iteratively improve the solution (in terms of the reward objective), jumping from one solution to another.

NeuRewriter is applied to three NP-Hard problems: Vehicle Routing Problem (VRP), Halide Expressions Semplification (HES) and Multi-Resources Tasks (MRT), using the same architecture for all three problems, but training separate models.
We will concentrate on the application of NeuRewriter to MRT, given that MRT is a variant of JSSP, but we will also see the results NeuRewriter on HES and VRP.

4.1 Search space

In the repairing approach, the search space for an instance of a NP-Hard problem is the set of all solutions for that instance. Differently from the constructive approach, NeuRewriter makes assumptions on the structure of the search space, by defining a search space as **suitable** [5] if it has 2 characteristics:

1. A feasible solution is easy to get. For example we can use Earliest Job First (EJF) for the MRT problem (P complexity), or Most Work Remaining (MWKR) for JSS problem (P complexity).

2. The search space should have "well-behaved local structures" [5], which means that it is possible to find some policy $\pi(a|s)$, such that we can iteratively jump from a solution s to a better solution s'. More in general, if a search space is such that the good policies for that problem have to make a lot of bad decisions that gives a small negative reward, and a few good decisions that gives a large reward, then the search space does not have "well-behaved local structures".

4.1.1 Multi-Resources Tasks Problem

Suppose we have a single machine with m Jobs $J_1, \ldots, J_m$, each of which is a tuple $J_i = (\{D_1, \ldots, D_k\}, A_i, T_i)$, and where $D_j \in [0, 1]$ are the resources, A_i is the arrival time, and T_i is the duration of the job.
In addition, we define B_i as the time in which the job J_i is taken, and $C_i = B_i + T_i$ as the completion time [5] (An example in Figure 5).
The objective of the MRT problem is to find a schedule for the m jobs in a single machine such that the average waiting time is minimized, and such that the schedule satisfies the following constraints.

- A Job J_i can be taken only from it's arrival time A_i onwards.

- At any timestep t, the sum of the resources of the scheduled jobs, for each resource type, can't exceed 1 (100%).

- When the execution of a job starts, it must go on until it finishes.

Each solution for an instance of the MRT problem has one univoque DAG $G(V, E)$ encoding defined as follows.

1. There are $|J| + 1$ nodes, where v_0 represents the machine, and v_i represents J_i: $V = \{v_0, v_1, \ldots, v_{|J|+1}\}$.

2. An edge (v_0, v_i) is added when J_i is taken as soon as it becomes available ($A_i = B_i$), and an edge $(v_{i'}, v_i)$ is added when J_i is taken as soon as $J_{i'}$ finishes ($B_i = C_{i'}$).

We provide as example both a solution and its DAG encoding in Figure 5, to the following MRT instance, $J_1 = \{D_1 = (0.5, 0.4), A_1 = 1, T_1 = 2\}, J_2 = \{D_2 = (0.5, 0.2), A_2 = 2, T_2 = 2\}, J_3 = \{D_3 = (0.2, 0.4), A_3 = 2, T_3 = 1\}, J_4 = \{D_4 = (0.3, 0.4), A_4 = 1, T_4 = 3\}, J_5 = \{D_5 = (0.5, 0.5), A_5 = 3, T_5 = 1\}$.

In the original work, NeuRewriter solves the online version of MRT, which redefines the problem, by adding a queue and a new constraint as defined below.
In the online version of MRT there are m jobs, and a queue W containing at most $|W|$ jobs, with jobs arriving in the queue at each timestep with some probability, with the additional constraint that if the queue is full, and a new job J_i arrives, then, either J_i or a job in the queue must be scheduled immediately.

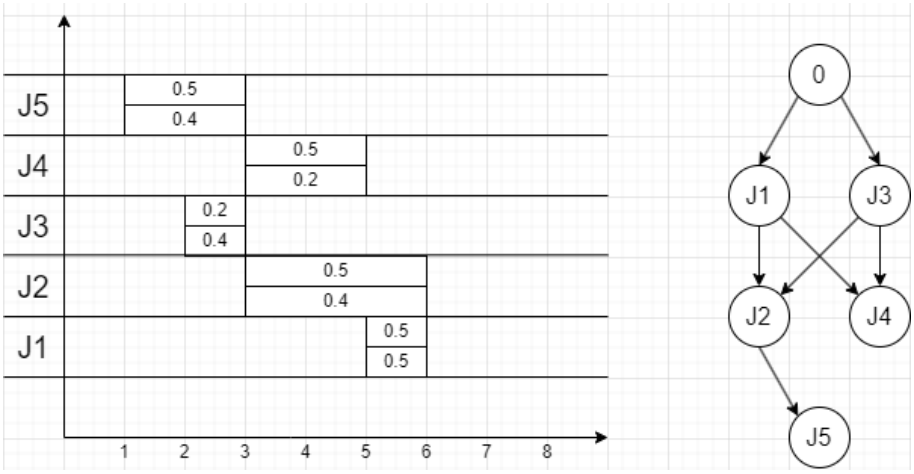

Figure 5: On the left a solution of the aformentioned MRT instance with 5 Jobs and 2 Resources, and on the right the DAG for that solution. Note that at any time t, the sum over one resource type is always ≤ 1.

4.1.2 States and embeddings

The states of the MRT problem are represented as $s = ((J_1, \ldots, J_m), \text{DAG})$, where $(J_1, \ldots, J_m)$ are the scheduled jobs, and DAG is the scheduling order of the solution, computed as shown in the previous section 4.

A solution could be embedded simply using the scheduled jobs, but the authors of NeuRewriter decided to make use of the scheduling order DAG, in order to make the scheduling order more explicit in the embedding.

In order to make use of the region and rule-secting policies, which are neural networks, we first need to turn the solution s=($(J_1, \ldots, J_m)$, DAG), into a set of vector embeddings. To do this we first create m embeddings $e_1, \ldots, e_m$ where each embedding is a vector $e_i \in R^{D*(T_{max}+1)+1}$ that represents the scheduled job J_i, and where T_{max} is a constant representing the maximum possible duration of a job (implementation details can be found in NeuRewriter paper [5]). Then we make use of an extended version of Child-Sum Tree LSTM architecture [9], in order to embed both e_i and the DAG in one embedding h_i:
Suppose the scheduled instance J_i, which is represented as v_i in the DAG has p parents: $parent(v_i) = \{v_k, v_{k+1}, \ldots, v_{k+p-1}\}, p = |parent(v_i)|$, then

$$(h_i, c_i) = LSTM[(\sum_{j \in parent(v_i)} (h_j), \sum_{j \in parent(v_i)} (c_j)), e_i]$$

where h_0, c_0 are randomly initialized, h_j,c_j are the embeddings of the node v_j.

By computing the embeddings for each e_i, we obtain an array of m embeddings: $h_1, \ldots, h_m$, univoquely representing a solution.

4.1.3 Actions

The set of rewriting rules is to reschedule a job v_i and allocate it after another job $v_{i'}$ finishes or at its arrival time A_i.
Given a queue of W jobs in the online MRT, the rewriting rules set has size $2 * W$, since each job could only switch its scheduling order with at most W of its former or latter jobs. [5]

4.1.4 Reward Function: Average Waiting Time

The reward function for all the 3 problems solved by NeuRewriter is defined as

$$r(s_i, a_i = (w_i, u_i)) = c(s_i) - c(s_{i+1})$$

where c is an evaluation function which is defined specifically for each different problem. In MRT, we define the Average Waiting Time function (or Average Job Slowdown) as the evaluation function: $c(s) = \frac{1}{k} * \sum_{i=1}^{m} (\frac{C_{s[i]} - A_{s[i]}}{T_{s[i]}})$, where $s[i] = J_i$ of solution s.
The Average Waiting Time function measures how much time in average, jobs have to wait from the time they become availabe A_i to the time in which they are taken B_i. When all jobs are taken as soon as they arrive ($B_i = A_i$), then $c(s) = 1$, else, as $c(s)$ becomes larger, the quality of the solution decreases. An example is provided in figure 6

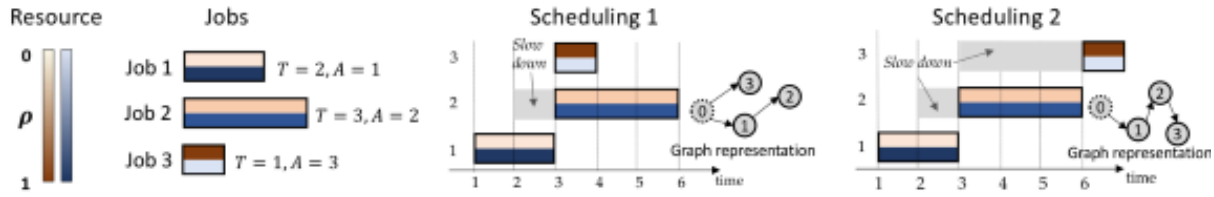

Figure 6: An example to illustrate the evaluation function for MRT [5]. On the center, scheduling1 has an evaluation value of 10/9, while scheduling2 has an evaluation value of 19/9. Scheduling 1 is clearly better than scheduling2, and if we were to jump from scheduling1 to scheduling2, the reward function $r(s_i, a_i = (w_i, u_i))$ would be negative.

4.2 Model and Architecture

NeuRewriter solves NP-Hard problems by first selecting a region w_t of the solution using a region-selecting policy $w_t \sim \pi_\theta(w_t|s_t)$, and then selecting a rule u_t using a rule-selecting policy $u_t \sim \pi_\theta(u_t|s_t[w_t])$ [5].

In the case of MRT, a region is an embedding $w_t = h_i$, that contains information about the job J_i along with its DAG scheduling order, while selecting a rewriting rule means to choose a couple of embeddings $(h_i, h_{i'})$, that are linked by an edge in the DAG of a solution s_t, and swap them.
See Figure 7 for examples of regions and rules applied to the NP-Hard problems in NeuRewriter.

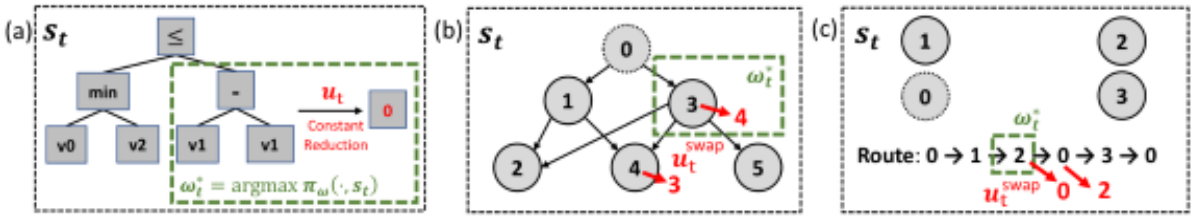

Figure 7: This example shows the regions and rules for a) Halide Expression semplification, b) MRT and c) VRP. We can see that in b) the region w_t is a node, while the rule u_t selects and swaps two nodes [5].

4.2.1 Region-Selecting Policy

The region selecting policy $\pi_w(w_t|s_t)$ outputs the probability distribution of regions, which is computed by applying a score function $Q_\theta(s_t, w_t)$ to every region of the state s_t, followed by a Softmax function.
We define the region-selecting policy as

$$\pi_w(w_t|s_t) = \frac{exp(Q_\theta(s_t, w_t))}{\sum_{w_i \in s_t} exp(Q_\theta(s_t, w_i))}$$

and proceed to learn a score function $Q_\theta(s_t, w_t)$, which computes a score for every region of the solution, where a high score indicates that rewriting the region w_t in the solution s_t: $s_t[w_t]$, could be desiderable [5].
In order to learn $Q_\theta(s_t, w_t)$, we fit it to the cumulative reward sampled from the current policies π_w, π_u:

Given an episode $(s_0, (w_0, u_0)), \ldots, (s_{T-1}, (w_{T-1}, u_{T-1})), s_T$, the regions-selecting loss $L_w(\theta)$ is defined as follows:

$$L_w(\theta) = \frac{1}{T} \sum_{t=0}^{T-1} \left(\sum_{t'=t}^{T-1} \left(\gamma^{t'-t} r(s_{t'}, (w_{t'}, u_{t'})) \right) - Q_\theta(s_t, w_t) \right)^2$$

where $Q_\theta(s_t, w_t)$ is implemented as a forward dense neural network with 1 hidden layer with ReLu activation and 256 neurons, receiving as input the embedding of a region h_i, and giving as output a score c_i.
After having applied $Q_\theta(s_t, w_t)$ to all the regions of the solution s_t, we have an array of scores $(c_1, \ldots, c_m)$, where m is the number of regions, and in the case of MRT, $m = |J|$. Softmax and multinomial sampling can then be applied to $(c_1, \ldots, c_m)$ in order to select a region w_t.

4.2.2 Rule-Selecting Policy

Given the region of the solution we want to rewrite $s_t[w_t]$, the rule-selecting policy outputs the probability distribution of the rewriting rules that can be taken in region $s_t[w_t]$: $\pi_u(u_t|s_t[w_t])$, from which a rule u_t can be sampled.

The rule-selecting policy is implemented using a REINFORCE+Baseline method [8], which reduces variance and speeds up learning with respect to Vanilla Policy Gradient using the learned $Q_\theta(s_t, w_t)$ function as baseline.
Given an episode $(s_0, (w_0, u_0)), \ldots, (s_{T-1}, (w_{T-1}, u_{T-1})), s_T$, the rule-selecting loss $L_u(\phi)$ is defined as follows:

$$L_u(\phi) = -\sum_{t=1}^{T} \left(\sum_{t'=t}^{T} \left(\gamma^{t'-t} * r(s_{t'}, (w_{t'}, u_{t'})) \right) - Q_\theta(s_t, w_t) \right) \cdot ln(\pi_u(u_t|s_t[u_t], \phi))$$

The architecture of the rule-selecting policy is split in two separate Multi-Layer Perceptrons (MLP), the first of which, given a regions v_i creates a vector of embeddings of all possible rewriting rules that can be taken from that region $\{h'_{i_1}, \ldots, h'_{i_{|U|}}\}$, and the second uses the output of the first, to compute the probability distribution of the rewriting rules U.

Given the region v_i (a node in the DAG in the case of MRT) with nodes $\{v_{i_k}\}$, being the set of nodes that are the parents of v_i, a set of couples embeddings is created coupling

v_i with each of its parents v_{i_k}, such that the size of the set $K = |\{(h_i, h_{i_k}\})|$ is less or equal to the set of all possible rewriting rules U that can be taken from a region: $K \leq |U|$.

Each couple of embeddings, then, is concatenated into a single vector (in the case of MRT, $(h_i, h_{i_k}) \in R^{1024}$), and fed to a MLP with one hidden layer with 256 neurons and ReLu activation, and outputs an embedding $h'_{j_k} \in R^{512}$, which represents the couple (h_i, h_{i_k}).
The set of K couple embeddings $\{h'_{i_k}\}$ is computed, and if $K < |U|$, then the embeddings $h'_{i_{K+1}}, h'_{i_{K+2}}, \ldots, h'_{i_{|U|}}$ are set to vectors of zeros (of size R^{512}).

Given the set of couple embeddings $\{h'_{i_k}\}$ computed above, all of the couple embeddings are concatenated into a vector $(h'_{j_1}, \ldots, h'_{j_{|U|}}) \in R^{|U|*512}$, and fed to a MLP with one hidden layer with 256 neurons and ReLu activation, with the output layer having $|U|$ neurons, and softmax activation.
In the end we get an array of $|U|$ probabilities $p_1, \ldots, p_{|U|}$, which is the probability distribution of the rewriting rules, from which we can sample a rewriting rule u_t.

4.2.3 Rewriting a solution for the MRT problem

Given a solution s_t, we described a method to sample both a region $w_t = v_t$ (the job to reschedule), and an action $u_t = (v_t, v_{t'})$ (a couple of jobs), but simply swapping the two jobs could result in violating the resource of time constraints of the other scheduled jobs, for this reason we need a function $s_{t+1} = f(s_t, v_t, u_t)$, that reschedules the jobs that are violating some constraint after the swap (for example, given a job v_1 with a duration of 3, a job v_2 with a duration of 4, and a job v_3 that is scheduled right after v_1 in s_t, then after swapping v_1 with v_2, v_3 will begin before v_2 finishes, and the resources may not be sufficient to allow that. To fix this we need to reschedule v_3, which was not involved directly in the swap, but suffered the consequences), see Figure 8.

Algorithm 1 Algorithm of a Single Rewriting Step for Job Scheduling Problem

function REWRITE($v_j, v_{j'}, s_t$)
  if $C_{j'} < A_j$ or $C_{j'} == B_j$ **then**
    return s_t
  end if
  if $j' \neq 0$ **then** $B'_j = C_{j'}$ **else** $B'_j = A_j$ **fi**
  $C'_j = B'_j + T_j$

  //Resolve potential resource occupation overflow within $[B'_j, C'_j]$
  J = all jobs in s_t except v_j that are scheduled within $[B'_j, C'_j]$
  Sort J in the topological order
  for $v_i \in J$ **do**
    B'_i = the earliest time that job v_i can be scheduled
    $C'_i = B'_i + T_i$
  end for
  For $v_i \notin J$, $B'_i = B_i$, $C'_i = C_i$
  $s_{t+1} = \{(B'_i, C'_i)\}$
  return s_{t+1}
end function

Figure 8: This example shows the algorithm for the rewriting rule for the MRT problem [5]. Given v_j, $v_{j'}$ that were chosen with the region-selecting and rule-selecting policies, this algorithm first checks if the jobs can be swapped, and then it swaps them, by rescheduling in topological ordering the jobs $v_i \neq v_{j'}$ that falls on the time interval $[B_{j'}, C_{j'}]$, of the swapped jobs $x_{j'}$, leaving the rest of the jobs untouched.

4.3 Training

The overrall objective function that is trained combines both the Region-Selecting loss $L_w(\theta)$ and the Rule-Selecting policy loss $L_u(\phi)$ into a single loss

$$L(\theta, \phi) = L_w(\theta) + \alpha * L_u(\phi)$$

where α is an hyper-parameter that is set to 10 for all the problems on which NeuRewriter is trained.

The algorithm presented in Figure 9 presents the details of the forward pass during training.

Algorithm 2 Forward Pass Algorithm for the Neural Rewriter during Training

Require: initial state s_0, hyper-parameters $\epsilon, p_c, T_{iter}, T_\omega, T_u$

for $t = 0 \rightarrow T_{iter} - 1$ **do**
  for $i = 1 \rightarrow T_\omega$ **do**
    Sample $\omega_t \sim \pi_\omega(\omega_t|s_t;\theta)$, where $\omega_t \in \Omega(s_t)$, $\pi_\omega(\omega_t|s_t;\theta) = \frac{\exp(Q(s_t,\omega_t;\theta))}{\sum_{\omega_t}\exp(Q(s_t,\omega_t;\theta))}$
    if $Q(s_t, \omega_t; \theta) < \epsilon$ **then**
      Re-sample $\omega'_t \sim \pi_\omega(\omega'_t|s_t;\theta)$ with a probability of $1 - p_c$
      if Re-sampling is not performed **then break fi**
    else
      break
    end if
  end for
  for $i = 1 \rightarrow T_u$ **do**
    Sample $u_t \sim \pi_u(u_t|s_t[\omega_t];\phi)$
    if u_t can be applied to $s_t[\omega_t]$ **then break fi**
  end for
  if u_t does not applied to $s_t[\omega_t]$ **then break fi**
  $s_{t+1} = f(s_t, \omega_t, u_t)$
end for

Figure 9: This example presents the forward pass algorithm for Neural Rewriter during training for an episode of size T_{iter}[5]. At every iteration a region w_t and a rule u_t are sampled from the current policies, w_t is resampled if it has a negative Q-Value, while u_t is resampled if it is not applicable. In the case of MRT, the hyperparameters are $T_{iter} = 50$ (size of the episode), $T_u = T_w = 10$ (maximum number of re-samplings), $\epsilon = 0.0$ and $p_c = 0.5$ is decayed by 0.8 every 1000 steps, until $p_c = 0.01$. The function f is the "Rewriting Step Algorithm", shown in section 4.2.3.

The dataset D, which contains 100K job sequences divided in 80K,10K,10K as train/-validation/test set, was randomly generated as shown below. NeuRewriter was trained to solve the online MRT problem, with every sequence containing 50 jobs and with a queue W of size 10, where jobs arrives on the fly.
The maximal duration of a sequence is $T_{max} = 50$ and the latest arrival time for a job $A_{max} = 50$ [5].

To test for generalizability and stability of NeuRewriter, the dataset D is generated by sampling from different distributions of job sequences, where each distribution defines a job property as a random variable, while keeping the other job properties fixed.
A NeuRewriter model is trained separately on each distribution.

1. **Number of resources type**: The number of resources D that each job in the sequence varies in the range $[2, 20]$. Average job arrival is steady, Resource distribution and Job lengths are non-uniform.

2. **Average job arrival**: The probability of a job arriving in the queue at timestep t can be either steady (70% of arrival), or dynamic (the percentage changes randomly). D=20, Resource distribution and Job lengths are non-uniform.

3. **Resource distribution**: The quantity of resources occupied for each resource type can be either uniform (for example, for D=2, both resources are set to 0.4), or non-uniform (for example, for D=2, the first resource is set to 0.3, while the second to 0.4) for each job of the sequence. D=20, Average job arrival is steady and Job lengths are non-uniform.

4. **Job lengths**: The duration of each job in the job sequence can be either long $[10, 15]$, or short $[1, 3]$. D=20, Average job arrival is steady and Resource distribution is non-uniform.

5 Analysis

In this section we first provide a comparison between the constructive and repairing approach to solve NP-Hard problems, in particular motivating the comparison between the results of NeuRewriter [5] on MRT and the results of the constructive approach [1] on JSSP. The application of the repairing approach to JSS is also provided as a research question, with details on how to do the actual implementation.

5.1 Comparing constructive and repairing approaches

Comparing NeuRewriter and the constructive approach is problematic because these two approaches do not solve the same problems: NeuRewriter solves Multi-Resources Task, VRP and Halide Expression semplification, while the constructive approach solves JSSP. In order to compare the models, we will use the results of NeuRewriter applied to MRT (which is a variant of JSS), and the results of the constructive approach on JSS, discarding the other confrontations because either they are similar problems that prioritize different strategies (JSS-VRP)[10], or they are very different problems (JSS-Halide).

5.1.1 Assumptions of the repairing approach

The repairing approach, differently from the constructive approach, makes two assumptions on the solution search space of a problem, such that if satisfied, would make repairing especially suitable to solve the problem.

1. An initial solution can be found very easily, with a polynomial-time algorithm (for example Shortest Job First for JSS).

2. The solution search space should have "well-behaved local structures", that is, it should be possible to find a policy such that, by jumping from a solution to another, the solution is being iteratively improved.

The repairing approach also formulates a **scalability hypothesis**, such that if the above assumptions hold, then, as the instance of an NP-Hard problem gets larger, the repairing approach will obtain better solutions than constructive approaches.

The authors of NeuRewriter [5] justify this hypothesis saying that, as the instance of a problem becomes larger, repairing a solution that was already computed with a euristic such as SJF (Shortest Job First), gives the repairing algorithm some basic knowledge of a solution that is already quite good, and just needs some corrections, while the constructive approach would have to do everything from scratch.

For the purpose of this work, that is to formulate an application of the repairing method to JSS, we will assume that JSS solution search space also satifies the two assumptions (the first assumption is trivial to check, while for the second we hypothesize that the good performances of NeuRewriter on a variant of JSS (MRT) means that also JSS has "well-behaved local structures").

5.1.2 Comparison of constructive and repairing approaches

The constructive approach trained on JSSP [1] was tested against two other constructive approaches in the literature: (Zhang et al. 2020) and (Hang and Yang 2020), two euristics: Most Work Remaining (MWKR) and First-in First-out (FIFO), and OR Tools [4]. The results are shown in Figure 10. The constructive method outperforms both the euristics, and the other constructive approaches found in literature, but is outperformed by OR Tools, which scores 7% better on Taillard's instances, and 6% better on Demirkol's instances than the constructive approach [1].

Dataset	Instance	Ours	FIFO	MWKR	(Zhang et al. 2020)	(Han and Yang 2020)	OR Tools	Upper Bound
Taillard	ta41	**2208**	2543	2632	2667	2450	2144	2005
	ta42	**2168**	2578	2401	2664	2351	2071	1937
	ta43	**2086**	2506	2385	2431	—	1967	1846
	ta44	**2261**	2555	2532	2714	—	2094	1979
	ta45	**2227**	2565	2431	2637	—	2032	2000
	ta46	**2349**	2617	2485	2776	—	2129	2004
	ta47	**2101**	2508	2301	2476	—	1952	1889
	ta48	**2267**	2541	2350	2490	—	2091	1941
	ta49	**2154**	2550	2474	2556	—	2089	1961
	ta50	**2216**	2531	2496	2628	—	2010	1923
	Average	**2203**	**2549**	**2449**	**2604**	**—**	**2058**	**1948**
Demirkol	dmu16	**4188**	4934	4550	4953	4414	3903	3751
	dmu17	**4274**	5014	4874	5379	—	3960	3814
	dmu18	**4326**	4936	4792	5100	—	4073	3844
	dmu19	**4195**	4902	4842	4889	—	3922	3764
	dmu20	**4074**	4539	4500	4859	—	3913	3703
	Average	**4211**	**4865**	**4712**	**5036**	**—**	**3954**	**3775**

Figure 10

The repairing approach trained on Online MRT was tested against Euristics: Shortest

Job First (SJF), Earlier Job First (EJB) and Shortest First Search (SJFS), against offline planning methods which solved the offline version of the problem: SJF-offline and OR Tools, and against a constructive approach: DeepRM [11], which were all applied/trained on the dataset D generated in the NeuRewriter work, shown in section 4.3.

In Figure 11, we can see that NeuRewriter is able to generalize well between different probability distributions. The **scalability hypothesis** is also confirmed when NeuRewriter is compared with DeepRM (a constructive approach which uses REINFORCE policy-based algorithm [8]): as the number of resources D increases DeepRM solutions becomes increasingly worst, while the quality of the solutions of NeuRewriter is little changed.

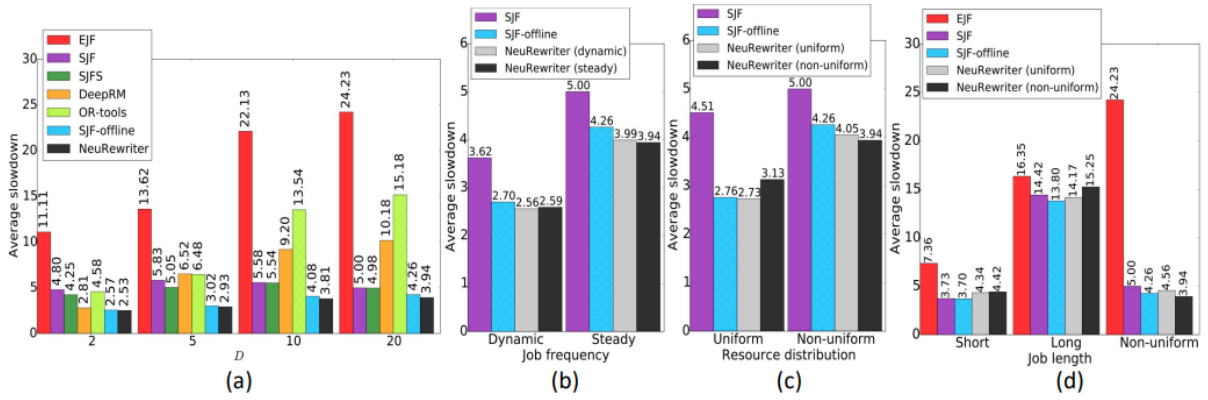

Figure 11: Experimental results of the MRT problem varying the following aspects: (a) the number of resource types D; (b) job frequency; (c) job frequency; (d) job length [5].

It is not possible to directly draw conclusion on which method is better without testing both on the same problem, namely JSSP, but the results of NeuRewriter on MRT, a similar problem to JSS, and the similarities between DeepRM and the constructive approach for JSSP (the first uses policy-based REINFORCE, while the second uses Actor-Critic PPO, but they both are constructive approaches), suggests that, if the hypothesis about the "suitability" of the repairing approach on JSS holds, then the repairing approach should behaves better than the constructive approach when applied to JSS.

6 Applying the repairing method to JSSP

In order to investigate the hypothesis that the repairing approach behaves better than the constructive approach on JSSP we discuss a possible implementation, where we apply the repairing approach to JSSP.

In the following section we link JSS solutions to Disjunctive Graphs and provide a possible implementation of the repairing approach to JSS, along with some research questions.

6.1 The Disjunctive Graph

The Disjunctive graph is a Directed Acyclic Graph (DAG) $G = (V, C \cup D)$ which allows to represent all **feasible solutions** of JSSP instances.
More formally, the Disjunctive graph is defined as $G = (V, C \cup D)$ [2], see Figure 12.

- V is the set of vertices corresponding to operations, plus two nodes representing the start and end of the scheduling. Each vertex, except the start and end ones, are operations of a job.
- C is the set of **conjunctive edge**, that contains edges between the i^{th} and $(i+1)^{th}$ operation of a job J, and represents an order of execution constraint. We also add conjunctive arcs from s to every first operation of every job, and from every last operation of every job to t. Black arcs in Figure 12.
- D is the set of **disjunctive edge**, where each edge links operations that are to be processed on the same machine, and represents an order of execution constraint. Dotter arcs in Figure 12.

To determine a schedule we have to define an ordering of all operations processed on each machine. This can be done by orienting all dotted or dashed edges such that each clique corresponding to a machine becomes acyclic.

We also want to avoid cycles between disjunctive and conjunctive arcs because they lead to infeasible schedules. A feasible schedule is represented by a directed acyclic disjunctive graph and a complete orientation of the edges in D defines a feasible schedule if and only if the resulting directed disjunctive graph is acyclic [2].

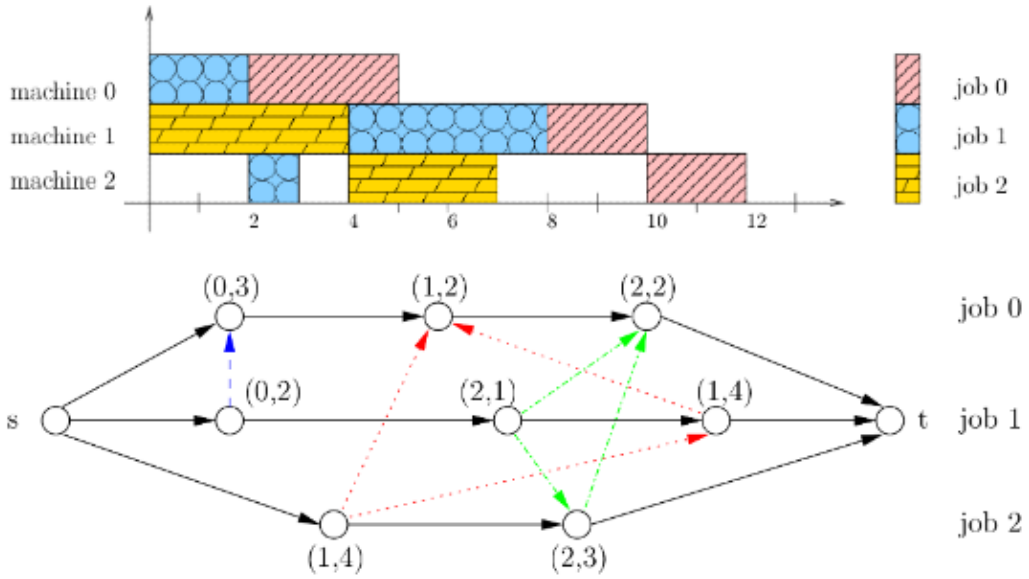

Figure 12: An example of a disjunctive graph (below), representing a specific solution (above) of the JSSP instance: $\{J_0$=[(0,3),(1,2),(2,2)], J_1=[(0,2),(1,2),(1,4)], J_2=[(1,4),(2,3)]$\}$.

An important property of Disjunctive Graph of which we will make use on the next section is that, starting from any given feasible solution s_0 of a JSSP instance, it is possible to reach any other feasible solution s_j of the same instance in a finite number of steps, where a step is defined as reversing the direction of a disjunctive edge.

6.2 The Repairing Approach applied to JSSP

In the last section we introduced Disjunctive Graphs , and we will make use of them by defining the search space of the repairing approach applied to JSSP as the set of all feasible solutions of a given JSSP instance.

There are a couple of different ways to define what a region, and what a rule are in JSSP:

1. A region is a disjunctive edge $(v_j, v_{j'})$, and, given that region, the only possible rule is to change the direction of the selected edge.

2. A region is a single node v_j of the Disjunctive graph G (this is the same encoding that NeuRewriter applied to MRT), and the rules are the couple of nodes $(v_j, v_{j'}) \in G$, such that $(v_j, v_{j'})$ is a disjunctive edge.

3. A region is a subgraph $C \subset G$, that contains all the nodes and edges in G that are scheduled in the same machine M_i, and the rules are the couple of nodes $(v_j, v_{j'}) \in C$, such that $(v_j, v_{j'})$ are disjunctive edges.

In this section we provide the implementation of the first of the three options, and leave as an open research question the discussion and implementation of the others two.

6.2.1 States and embeddings

To solve JSSP, we define a region-policy $\pi_w(w_t|s_t;\theta)$, where the set of all possible regions of a solution: $\Omega(s_t)$ is defined as the set of disjunctive arcs D_{s_t} of the solution s_t.
Differently from NeuRewriter, that in order to jump from a solution s_t to another solution s_{t+1}, defined both a region-selecting and a rule-selecting policy, we will embed both of them into a region-rule-selecting policy $\pi_u(u_t|s_t;\theta)$ because, once an edge u_t is sampled, there is only one possible action (to change the direction of the edge).

As with NeuRewriter, states needs to be embedded into vectors in order to be fed to the policy neural network.
In our case a region/rule is a disjunctive arc $(v_i, v_{i'})$, so we first create an embedding e_i for every node v_i in G, using a function $e_i = f(v_i)$, as it was done in NeuRewriter 4.1.2. The function f should create an embedding which uniquely identifies the operation v_i, here we define a naive function f, but a better implementation is left as an open research question.

Given a node v_i, representing the operation $o_{v,j} = \{d_{v,j}, m_{v,j}\}$ of the job J_v in the disjunctive graph G, scheduled at timestep t, we define e_i as the following vector: $e_i = (v_i, d_{v,j}, m_{v,j}, t) \in R^4$.

Given the set of embeddings $e_1, \ldots, e_K$, where K is the number of nodes in the disjunctive graph, we proceed in the same way NeuRewriter did, by using the same Child-Sum Tree LSTM architecture as NeuroRewriter [9] (See Section 4.1.2), with the Disjunctive Graph as DAG, in order to embed both e_i and the Disjunctive Graph information into one embedding $h_i \in R^{512}$: Suppose the scheduled operation $o_{i,j}$, which is represented as v_i in the DAG has p parents: $parent(v_i) = \{v_k, v_{k+1}, \ldots, v_{k+p-1}\}, p = |parent(v_i)|$, then

$$(h_i, c_i) = LSTM[(\sum_{j \in parent(v_i)} (h_j), \sum_{j \in parent(v_i)} (c_j)), e_i]$$

where h_0, c_0 are randomly initialized, h_j,c_j are the embeddings of the node v_j.

6.2.2 Reward Function

We use the makespan as reward function. Given that our search space only contains feasible solutions, makespan is a **dense reward function**.

6.2.3 Objective Function and Architecture

As stated in the Disjunctive Graph section 6.2, using Disjunctive Graphs enables the usage of an algorithm capable of reaching any feasible solution $s_{j'}$, starting from any other feasible solution s_j.

Given that, once $u_t = (v_j, v_{j'})$ is sampled from the region-rule-policy $\pi_u(u_t|s_t;\theta)$, there is only one possible rewriting rule u_t (to change the direction of the selected disjunctive edge u_t), we decided to drop the rule-policy $\pi_u(u_t|s_t[w_t];\phi)$ which was present in NeuRewriter.

A problem of removing rule-policy from NeuRewriter is to decide which model the new region-rule-policy should use.
If we kept NeuRewriter region-policy we would be learning a value function $Q(w_t, s_t;\theta)$, and then computing the policy with the Softmax function, but the usage of a complete rollout as an unbiased estimator for learning $Q(w_t, s_t;\theta)$ causes high variance [1] (See Figure 4.2.1 for the region-selecting loss of NeuRewriter).
Alternatively, we propose to use an actor-critic Proximal Policy Optimization method (AC-PPO) instead [7], the same model and architecture as the constructive approach paper used [1], in order to get both a more stable learning and to avoid performance collapse.

We split the region-rule-selecting policy in two networks, as NeuroRewriter did with the rule-selecting policy 4.2.2.

The first network is a MLP with one hidden layer with 256 neurons and ReLu activation (the same as NeuroRewriter), that takes as input $(h_i, h_{i'})$ (the embeddings of the disjunctive edge $(v_i, v_{i'})$), and outputs an embedding $h_i^{'} \in R^{512}$. As with NeuRewriter, we end up with a set of region embeddings $h_1^{'}, \ldots, h_K^{'}$.

The second network is the same architecture as the one presented in Figure 4, with the following differences: (1) The input is the vector $(h_1^{'}, \ldots, h_K^{'}) \in R^{512*K}$ instead of a $|J| \times 7$ matrix, (2) the function f, and the vector B are not present in the action-selection network, (3) the output of the action-selection network has size R^K, where K is the maximum number of operations possible (If we trained our model in instances of JSSP with 30 jobs and 20 machines, then the size of the output of the action-selection network would be $30 * 20 = 600$).

As we are using the same PPO method as the constructive approach, we define the objective function of the region-selecting policy the same way (See section 3.2).

6.2.4 Rewriting Step Algorithm for JSSP

We describe a rewriting step algorithm, that, given a rewriting rule u_t, computes $s_{t+1} = rewriting(s_t, u_t = (v_j, v_{j'}))$ (See Figure 6.2.4).
The algorithm tries to change the direction of the selected disjunctive edge $(v_j, v_{j'})$, but

only if the resulting Disjunctive Graph $s_{t'}$ is still acyclic (checking acyclicity requires O(V+E) time) the solution s_{t+1} is updated, else it remains the same.

Algorithm 1 Algorithm of a single rewriting step for JSSP

Require: s_t disjunctive graph, $(v_j, v_{j'})$ a disjunctive arc of s_t

procedure REWRITE($v_j, v_{j'}, s_t$)
    $s_{t'} = \text{swap}(v_j, v_{j'}, s_t)$ ▷ the directed edge $(v_j, v_{j'})$ is substituted with $(v_{j'}, v_j)$
    if check_cycle_dfs($s_{t'}$) **then** ▷ If $s_{t'}$ is cyclic, the solution is unfeasible
        Return s_t
    Return $s_{t'}$

6.2.5 Training

We redefine the Forward Pass Algorithm which was used to train NeuRewriter, see Figure 9 for NeuRewriter forward pass, by removing the rule-selector $\pi_u(u_t|s_t[w_t];\phi)$ and explicitely embedding the rewriting rule for JSSP, see Algorithm 6.2.5.

Algorithm 2 Forward Pass Algorithm for the Repairing Approach, applied to JSSP during Training, for a single episode of size T_{iter}

Require: initial state s_0, hyperparameters T_{iter}, T_u

for $t = 0 \rightarrow T_{iter} - 1$ **do**
    for $i = 1 \rightarrow T_u$ **do**
        Sample $u_t = (v_j, v_{j'}) \sim \pi_u(w_t|s_t;\theta)$ where $u_t \in \Omega(s_t)$
        $s_{t'} = \text{rewrite}(v_j, v_{j'}, s_t)$
        if check_cycle_dfs($s_{t'}$) is FALSE **then Break**
    $s_{t'} = \text{rewrite}(v_j, v_{j'}, s_t)$
    if check_cycle_dfs($s_{t'}$) is TRUE **then** ▷ No feasible region found, end the episode.
        Break
    $s_{t+1} = s_{t'}$

7 Conclusions

In this work we analyze both a constructive [1], and a repairing approach [5] to solve NP-Hard problems, and compare them. We find that the repairing approach obtained better results on the NP-Hard problems it solved when compared to the results obtained by the constructive approach on JSSP, and we propose a specific implementation of a repairing RL+DL network to solve JSSP, based on both papers studied in this work [1][5].

As for future work, in order to verify the **scalability hypothesis** of NeuRewriter applied to JSSP, either the proposed implementation, or an alternative one that is only

mentioned in section 6.2 should be implemented, and compared with the results in the constructive approach [1] on Taillard's and Demirkol's instances [12]. We expects the **scalability hypothesis** to hold true, and the repaired-based method to outperform the constructive-based one.

Finally, the function f used to embed the nodes of the Disjunctive Graph in section 6.2, and the proposed repaired-based architecture for JSSP, could also be investigated, and compared with the original ones used in NeuRewriter.

8 Bibliography

References

[1] A Reinforcement Learning Environment For Job-Shop Scheduling

[2] The Job-Shop Problem, the disjunctive model and benchmark data

[3] The Job-Shop Problem solved with TSP and Genetic Algorithms

[4] Google OR Tools

[5] Learning to perform local rewriting in combinatorial optimization

[6] Rigorous Analysis of Heuristics for NP-hard Problems

[7] Proximal Policy Optimization Algorithms

[8] Reinforcement Learning: An Introduction - Andrew Barton and Richard S. Sutton

[9] Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks

[10] Vehicle Routing and Job Shop Scheduling: What's the difference?

[11] Resource Management with Deep Reinforcement Learning

[12] Prof. Éric Taillard

[13] Wikipedia - JSSP