A Neural Approach to Generation of Constructive Heuristics

Mohamad Alissa
School of Computing
Edinburgh Napier University
Edinburgh, UK
M.Alissa@napier.ac.uk
ORCID: 0000-0002-9548-863X

Kevin Sim
School of Computing
Edinburgh Napier University
Edinburgh, UK
K.Sim@napier.ac.uk

Emma Hart
School of Computing
Edinburgh Napier University
Edinburgh, UK
E.Hart@napier.ac.uk

***Abstract*—Both algorithm-selection methods and hyper-heuristic methods rely on a pool of complementary heuristics. Improving the pool with new heuristics can improve performance, however, designing new heuristics can be challenging. Methods such as genetic programming have proved successful in automating this process in the past. Typically, these make use of problem state-information and existing heuristics as components. Here we propose a novel *neural* approach for generating constructive heuristics, in which a neural network acts as a heuristic by generating decisions. We evaluate two architectures, an Encoder-Decoder LSTM and a Feed-Forward Neural Network. Both are trained using the *decisions* output from existing heuristics on a large set of instances. We consider streaming instances of bin-packing problems in a continual stream that must be packed immediately in strict order and using a limited number of resources. We show that the new heuristics generated are capable of solving a subset of instances better than the well-known heuristics forming the original pool, and hence the overall value of the pool is improved w.r.t. both Falkenauer's performance metric and the number of bins used.**

***Index Terms*—Automatic Heuristics Generation, Hyper-Heuristics, Encoder-Decoder LSTM, Streaming Bin-packing.**

I. Introduction

Given a large set of problem instances in a combinatorial optimisation domain, it is well-known that the performance of any given heuristic will vary significantly from instance to instance. Hence, a pool of heuristics that have complementary performance within the instance space is likely to be beneficial. This performance complementarity can be exploited in multiple ways. For example, given a set of heuristics, an *algorithm-selector* can be used to choose the best heuristic for a particular instance. This task of automatically selecting a heuristic from a given set is known as the per-instance algorithm selection problem [1]. Alternatively, a *Hyper-Heuristic (HH)* [2] can combine low-level human-designed heuristics in a manner which allows the resulting method to outperform any of the individual heuristics when solving a combinatorial optimisation problem. Both approaches rely on a pool of quality heuristics. Adding new heuristics to the pool can improve performance across a set of instances, particularly if a new heuristic is diverse w.r.t the existing heuristics.

While heuristics can be selected from existing literature, the hyper-heuristic community have focused on new automated methods to generate new heuristics. The aim of *generative constructive HH* is to produce new low-level constructive heuristics, rather than designing them manually based on human intuition, which is a time-consuming and laborious process [3], [4]. Automating this process reduces the man-hours involved in deriving low-level heuristics and may lead to the induction of new constructive heuristics that humans would not think of [4]. Typical approaches to this use Genetic Programming (GP) [5] with its variations including: tree-based GP [3], [6]–[9], grammar-based GP [3], [10], gene expression programming [11], [12] and grammatical evolution [13]–[15]. Some studies have investigated other techniques for this purpose including, genetic algorithms [16], [17], single-node genetic programming [18] and artificial immune systems combined with genetic programming [19], [20].

Most of these methods rely on an existing pool of heuristics which become *components* of new heuristics, e.g. exist as nodes in a tree evolved using GP. We propose a novel approach in which we train neural network heuristics that learns from the *decisions* generated by a set of low-level heuristics — this is in stark contrast to typical methods for heuristic generation that combine low-level heuristics into new heuristics. We apply the approach to solving streaming bin-packing problems: domains which incorporate *streaming data* — data points that arrive in a continual stream, which may be large and potentially unbounded, and in which the order of data points cannot be influenced — which still pose considerable challenges for existing methods. Many real-world problems also have an additional constraint in that the items arriving in the stream have to be dealt with by limited resources, e.g. packing items from a conveyor into trucks in a holding area, or stacking problems (common in shipping and steel industries [21], [22]) in which large items are moved by crane between arrival, holding and delivery stacks, each with fixed capacity. Our method generates a heuristic that makes a decision per-item as it arrives in the stream[1].

978-1-7281-8393-0/21/$31.00 ©2021 IEEE

[1] An alternative would be to consider a sliding window and consider batches as distinct sub-problems

We train two types of neural model. The first uses a sequential-based architecture, namely an Encoder-Decoder LSTM[2]. This is compared to a classic Feed-Forward Neural Network (NN). These models take the size of the current item from a stream and the current state of open bins BS as input and output a decision: i.e. whether the item should be packed in a bin, or whether a bin should be closed and the item packed into a new one. Therefore, the trained models act as *constructive heuristics* in determining where an item is packed. The models are trained using *decisions* generated from each of a set of low-level heuristics using a large set of instances as training data.

We test the approach by evaluating the contribution of the new heuristics to a pool created by combining them with a set of existing well-known heuristics in the field of bin-packing on a large set of instances with increasing number of available bins. Specifically, we consider an online packing with streaming data in which items arrive one at a time, must be immediately packed and arrival items size are unknown in advance. The goal of the paper is to evaluate a new method for automated heuristic generation. The resulting pool can be used either with an algorithm-selector or a selective hyper-heuristic. Many methods exist for both the former and the latter and therefore we constrain our evaluation to the contribution of the heuristics to the pool. Results show that the generated heuristics are able to produce superior results to any of the heuristics used in training in 26%-31% of cases using a small number of available bins. Thus, expanding the baseline pool of three well-known heuristics from the literature with the new generated heuristics can obtain better overall performance.

The major contribution of the paper is to describe a novel approach to generating constructive heuristics using an Encoder-Decoder LSTM or NN that directly outputs decisions. The new method is rigorously evaluated on a streaming bin-packing problem by:

> Conducting an investigation of the scalability of the proposed approach with respect to the number of bins available for packing.
> Establishing a comparison of the new pools of heuristics that include Encoder-Decoder LSTM and/or a Feed-Forward Neural Network heuristics to the baseline pool of three constructive heuristics, in order to determine whether the new generated heuristics provide significant overall improvement.

As far as we are aware, this is the first time that such an approach has been used, and provides an alternative method for generating heuristics that can be used in algorithm-selection or selective hyper-heuristics. Although the method is evaluated in this instance on examples from the bin-packing domain, we expect that the proposed approach should easily generalise to other streaming domains such as the Block Relocation Problem (BRP) [21] or dynamic Job-Shop Scheduling [23].

[2]Long Short Term Memory

II. Related Work

Designing methods to generate new heuristics is an important sub-field of the hyper-heuristic community. A hyper-heuristic can be defined generally as "an automated methodology for selecting or generating heuristics to solve computational search problems [2]". Generation constructive HH have been successfully applied in many combinatorial optimization domains including scheduling [3], [6], [7], [20], bin-packing [16]–[19], [24]–[26], constraint satisfaction [10], vehicle routing [8], [14] and multidimensional knapsack problems [9]. These generated heuristics can be categorised as *disposable* where the heuristics are evolved for solving a single instance of a problem and not intended for using on unseen problems (Online learning HH), or *reusable* where the heuristics are generated for a set of training instances and they might generalise to unseen instances (Offline learning HH) [3], [4].

Real-world streaming problems (e.g. production lines) typically appear in a dynamic environment with stochastic events such as machine breakdowns and random job arrivals. This requires flexible responses to the changes in the conditions and constraints. The packing domain has attracted much previous attention: for example, generation constructive HH using tree-based GP approaches are particularly prevalent in the 1-D online BPP [24], [25], which take into account the size of the current item, the full capacity and the load of a bin. These attributes have been reduced to the size of the current item and the residual capacity of a bin (i.e. how much space is remaining in the bin) later in [26]. The research conducted by [25] shows that a simple GP tree can be used to discover human-designed heuristics such as first-fit by examining each bin in turn and place the item in the first suitable bin, while the research in [24], [26] evolved heuristics whose performance was comparable with best-fit by examining all of the bins and place the item in the bin which receives the maximum score. Also, the research in [24] sheds the light on the trade-off between the performance and generality of the generated heuristics and their robustness to new problems, where the choice of the training instances (categorised according to the item size distribution) is vital in the area of automatic heuristic generation.

Other techniques have also been employed to solve the 1-D online BPP. For example, genetic algorithms have been used to evolve low-level constructive heuristics in the form of a policy matrix [16], [17]. Depending on the residual space of the bin and the item size, a policy matrix indicates the weight for packing an item in a bin where the item is packed in the bin with the highest weight. This research shows that the generated heuristics are specialised to the distribution of item sizes and outperform the existing human-designed heuristics. Generation constructive HH are also applied successfully to solve offline BPP including [18], [19].

This paper presents a completely new approach to generating constructive heuristics using an Encoder-Decoder LSTM or a Feed-Forward Neural Network in the context of streaming 1-D BPP with limited resources (i.e. bins), trained using *de-*

cisions generated from low-level heuristics. Unlike generative HH approaches that search in the space of possible heuristics that may be suitable for solving a problem, the proposed approach navigates the space of possible decisions that create a solution for a given problem instance. This approach can handle an unbounded stream with stochastic arrival of individual items which are packed on arrival. It also examines all the available bins and uses dynamic information regarding bins-state to close full bins when necessary and open new ones if required. These trained neural models themselves essentially act as heuristics, directly outputting decisions. Thus, they are considered reusable heuristics that can be applied to new similar BPP instances with the same item size distribution.

We are of course aware that different methods for decision-making in a dynamic environment such as Monte Carlo Tree Search [27] are also likely to be able to solve the problem at hand. However, as described in section IV, the problem at hand is a sequence-to-sequence problem. Given the fact that Encoder-Decoder LSTM topologies have been applied successfully to solve sequencing problems in the Natural Language Processing field [28], [29], our driving motivation is to investigate whether they can also be used to generate heuristics to solve a combinatorial optimization problem.

III. Streaming Bin-Packing Data Instances

We consider the following streaming bin-packing/stacking problem scenario (Fig 1): *"Given a production line where the packing/stacking is carried out by a fixed robot arm/crane at the end of the line, the items (e.g. steel slabs) arrive one by one to be packed/stacked into a certain set of containers. When the item doesn't fit in any container, then the container with the lowest free space is closed and a new one is opened to pack/stack the item into."*

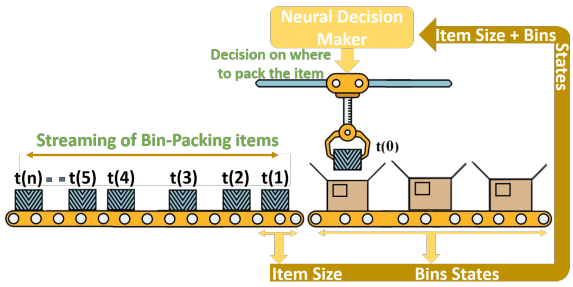

Fig. 1. The Packing Process

We use 900 bin-packing instances from datasets[3] first defined in [30], each of which has 120 items and is initialised with item sizes drawn from two different distributions. In order to solve this as a streaming instance, we consider the items to arrive in the order defined in each instance. From these instances, we define two balanced data sets DS(1,2) each with 450 instances as shown in table I as follows. Each instance is solved best by one of the heuristics under investigation (according to the Falkenauer fitness function [31] given in equation 1). The heuristics considered are — Best-Fit; First-Fit and Worst-Fit [30] (BF, FF and WF). Thus, 150 instances are solved best by each one of them.

$$Fitness = \frac{1}{b}\sum_{i=1}^{b}(\frac{fill_i}{c})^k \quad (1)$$

Where c = bin capacity which is fixed at 150, k is set to 2, $fill_i$ is the sum of the item sizes in bin_i and b is the number of bins used.

Each row of training data describes the *input* to the network as [*Item0*, $BinState_0$, $BinState_1$, $BinState_2$] and the related *output* as sequence of actions as [A_{bin0}; A_{bin1}; A_{bin2}]: a *bin state* is defined as the residual capacity of each bin (i.e. how much space is remaining in the bin) and an action A_{bin} specifies whether or not an item should be placed in a bin, or whether or not a bin should be closed and the item packed into a new one. This process of extracting this data is explained in the following steps:

1) Label each *instance* with the best performing heuristic, as described above.
2) Split the data set into 300 instances for training and 150 instances for testing purposes.
3) Concatenate all the training instances to create a long training stream of 36,000 items (300x120); this is repeated for the test set, resulting in a test stream with 18,000 items. Label each *item* with the heuristic H_{item} that best solved the instance the item came from.
4) For each item i in the training stream, determine the current bin states BS, and denote the input data as [*item*, BS].
5) Apply H_{item} which determines which bin the item will be placed in (assuming there are a fixed number of bins b available at any one time, each with bin capacity c = 150). Assign an action sequence A to the item as the desired output, e.g. [0,0,1] indicating the item is placed in the 3rd bin.

TABLE I
Data sets details [30]. Bin Capacity is fixed at 150

DS	total	n_{items}	Lower - Upper Bounds	Distribution
DS1	450	120	[40-60]	Gaussian
DS2	450	120	[20-100]	Uniform

IV. The Neural Approach For Decision Making

The proposed approach aims to make a sequence of actions per bin based on an item size and a sequence of current bins states, following which the item is packed accordingly, as presented in Fig 1. The decision is the non-zero action and this is checked for *validity* (e.g. to ensure an item can fit in a bin, explained in section V), then the approach packs the item and updates the current bin states. Dynamic information regarding the current bin states is fed back into the network to pack the next item in the stream (Fig 2).

[3]https://github.com/Kevin-Sim/BPP.

TABLE II
RANGE OF VALUES THAT USED IN THE ENCODER-DECODER LSTM HYPER-PARAMETERS TUNING; THE TABLE ALSO SHOWS THE FINAL SELECTED VALUES AND THE NEURAL NETWORK HYPER-PARAMETERS

	#Epoch	Batch Size	#Layer	Memory Units/Neurons	Optimiser	Loss Function
Tuning Range	100	[16, 2048]	[1_1 - 4_1, 2_2]	[32, 2048]	adam	categorical_crossentropy
Best-LSTM	100	128	1_1 + Full-Connect Layer	1024	adam	categorical_crossentropy
NN	600	128	4	(16)-(32)-(16)-(6 or 40)	adam	categorical_crossentropy

As previously noted in the literature [28], sequence-to-sequence (seq2seq) problems can be challenging since the input/output can have different lengths and come from different spaces. Encoder-Decoder LSTM topologies [28], [29] have been designed to deal with such problems and delivered outstanding results, particularly in Natural Language Processing field. As shown in Fig 2, an Encoder-Decoder LSTM topology comprises two main models: an **encoder** to summarise the input sequence into fixed vector and a **decoder** to predict a sequence based on the encoder output. This architecture uses a training technique called *teacher forcing* [32] as an effective and alternative to the normal BackPropagation Through Time (BPTT) for training RNNs architecture. A detailed explanation of the model is outwith the scope of this paper, the reader is referred to [28], [29] for a full explanation. As an additional neural approach, a non-sequential neural network [33] is employed, this is further described in the next section.

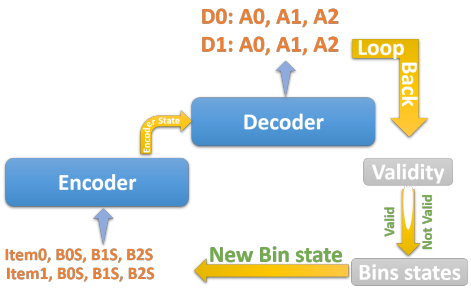

Fig. 2. The workflow of the Encoder-Decoder LSTM decisions making

V. METHODOLOGY

Keras functional API[4] is used for the LSTM implementation, where the input is a size of the item to be packed and a sequence of the current bin states, and the output is a sequence of actions per bin. The input and output are both one-hot encoded. A preliminary empirical investigation was conducted to tune the Encoder-Decoder LSTM architecture and hyper-parameters using the ranges shown in table II. The "Adam" optimiser [34] was used in all tuning experiments due to its reported accuracy, speed and low memory requirements. Using Keras, we also implement a classical neural network where the input/output are the same as in LSTM approach except that we do not one-hot encode them, and output actions (described below) are represented using two neurons (e.g. the last NN layer includes 6 neurons in the experiments with 3 bins and 40 neurons with 20 bins). The hyper-parameters of both the LSTM and NN are shown in table II. All experiments are conducted on Google Colab[5] with GPU run-time used to execute the experiments.

As previously described in section III, each dataset was split into a training set (67%) and test set (33%). These sets were created for two scenarios, each using a different number of bins (3 and 20). Each training set contains a list of input-output pairs in which the input is item size and list of the current bin states, and the output is a list of actions for each bin. Each experiment was repeated ten times, thus produces 20 different LSTM- and NN-generated heuristics. For each experiment, we save the model that provides the lowest error from the training phase then we test the trained models on the testsets. We consider three possible actions:

No packing in this bin (denoted by 0).
Pack the item in this bin (denoted by 1).
Close this bin, open a new one and pack the item in it (denoted by 2).

As mentioned before, the LSTM model outputs a sequence of actions per bin. We define the decision as the non-zero action; the item is packed into the bin indicated by the '1' in the action sequence. As the decoder predicts an action per bin recursively, the LSTM approach might output a sequence of conflicting actions (e.g. the item should be packed in multiple bins). In this case, we break the tie by applying the first non-zero action then correct all subsequent actions to 0 so that there is only one non-zero action. Despite the fact that using an architecture that can produce conflicting actions comes with a cost associated with their repair, in practice, this rarely happens: for instance, for DS2 using 20 bins only 1% of actions are conflicted per instance. NN approach predicts a decision directly (i.e. non-recursively) and thus only one non-zero action is output, i.e. there are no conflicting actions. Now after having a decision (the non-zero action), it is possible that the LSTM and NN can output an invalid decision, e.g. attempting to overfill a bin or not to place the item in any bin. We therefore apply a correction to these cases, ensuring a valid decision is always produced. These special cases and the corrections applied are listed in table III. It is worth noting that this technique of dealing with invalid decisions is common in the literature related to generating heuristics with constraints [16], [17], [24]–[26].

We determine the quality of the decisions made by considering a sequence of $n = 120$ decisions as a batch, and measuring the quality of the overall solution created for the

[4]https://github.com/fchollet/keras

[5]https://colab.research.google.com/notebooks/welcome.ipynb

TABLE III
SPECIAL CASES: CORRECTIONS APPLIED TO HANDLE CASES WITH INVALID DECISIONS OR MULTIPLE CONFLICTING ACTIONS WITH USING 3 BINS AS AN EXAMPLE

Sequence of Actions	Description	Type	Corrected Decision	Description
0,0,0	No packing action is provided	Invalid	2,0,0	Close the first bin and open new one to pack the item in
1,0,0 / 0,1,0 / 0,0,1	The bin-capacity is broken	Invalid	2,0,0 / 0,2,0 / 0,0,2	Close the chosen bin and open new one to pack the item in
1,1,0 / 1,1,1 / 0,1,1	Multiple packing actions	Conflicting	1,0,0 / 1,0,0 / 0,1,0	Pack the item in the first chosen bin
2,2,0 / 2,2,2 / 0,2,2	Multiple "open new bin" actions	Conflicting	2,0,0 / 2,0,0 / 0,2,0	Close the first chosen bin and open new one to pack the item in
2,1,0 / 2,1,1 / 2,1,2	Multiple mixed actions	Conflicting	2,0,0	Close the first chosen bin and open new one to pack the item in

entire batch as a result of applying each decision. we define the baseline oracle as the best of the three low-level heuristics (BF, FF and WF) used to create the training data. 'Best' is defined in terms of the Falkenauer fitness metric or number of used bins, depending on the experiment. Also, we define **Validity:** the percentage of decisions per batch that are *valid* in the sense that (1) there is at least one decision with value 1 or 2 (i.e. the item is packed) (2) no constraints regarding bin-capacity are broken.

VI. RESULTS

This section describes the experiments results as presented in tables IV to VIII. The section presents experiments to (A) Establish how often the Encoder-Decoder LSTM and Neural Network generated heuristics produce valid decisions; (B) Compare the generated heuristics performance to the individual constructive heuristics performance; and (C) Gain insight into how much benefit the generated heuristics bring to the baseline pool of heuristics.

A. Returning Valid Decisions

Tables IV shows that when using 3 bins both LSTM- and NN-generated heuristics obtain similar results in terms of generating valid decisions. It is clear that the NN heuristics fails to handle larger numbers of bins (20), generating many more invalid decisions. The poor performance of the standard neural network with the long sequences of bin states is not surprising, as a standard architecture cannot learn the mapping between the temporal information implicit in the bins states and the sequence of actions. Furthermore, the results show that the LSTM heuristics make valid decisions (that is, do not require correction for not packing an item or breaking bin-capacity) the majority of the time: [97%-99%] and [94%-99%] for DS1 and 2 respectively. The DS2 results in more invalid decisions than DS1. This is possibly due to a broader range of item sizes [20, 100] that are used to build DS2 instances (DS1 ranges from[40, 60]): the encoder may have more difficulty summarising instances with a wide range of values, potentially requiring more encoder layers for the model.

B. Comparing Neural-Generated Heuristics To Human-Designed Heuristics Performance

The oracles and their heuristics are presented in table V. The baseline **Oracle(1)** includes the human-designed constructive heuristics (BF, FF and WF); **Oracles(2) and (3)** expand the baseline oracle with the best-of-run (in terms of valid decisions) generated heuristic using LSTM and NN respectively;

TABLE IV
MEAN/STD OF THE VALIDITY RESULTS OBTAINED FROM 10 EXPERIMENTS OF LSTM AND NN APPROACHES ON 150 TESTING INSTANCES FROM DS1 AND DS2

#	LSTM	NN
DS1-3bins	99.87% (+/- 0.03%)	99.59% (+/- 0.26%)
DS1-20bins	97.79% (+/- 0.56%)	82.04% (+/- 1.95%)
DS2-3bins	99.37% (+/- 0.15%)	99.54% (+/- 0.25%)
DS2-20bins	94.14% (+/- 0.49%)	76.51% (+/- 3.19%)

TABLE V
THE HEURISTICS USED IN EACH ORACLE

	BF	FF	WF	LSTM	NN
Oracle(1)	✓	✓	✓	—	—
Oracle(2)	✓	✓	✓	✓	—
Oracle(3)	✓	✓	✓	—	✓
Oracle(4)	✓	✓	✓	✓	✓
Oracle(5)	✓	✓	✓	✓ 10	✓ 10

and **Oracle(4)** includes the best-of-run generated LSTM and NN heuristics. Tables VI shows the number of instances that are uniquely best solved using the oracles for DS(1,2) in terms of both Falkenauer's performance and number of bins used. The uniquely best heuristic is the heuristic that outperforms the other heuristics in a pool to solve a given instance, i.e. it has best performance and without being equal to any other heuristic performance. One would notice that the numbers of instances in table VI (especially in terms of number of bins) do not sum up to number of test instances (150 instances). Since we are only interested in the unique contribution of each heuristic, we do not present the number of instances that are solved equally using heuristics in a pool.

As we use balanced dataset in terms of Falkenauer's performance (described in section III), the heuristics in oracle(1) solve the same number of instances each in DS(1,2) however, these numbers can be different in terms of bins used. In the experiments using 3 bins from Falkenauer's performance perspective, the generated heuristics solve a good number of instances and sometimes exceed those solved by the well-known human-designed heuristic BF. The LSTM heuristic solves 31% of the test instances (47 instances) uniquely best outperforming BF that solves 29% (44 instances) in DS1. The LSTM heuristics are considered as first and second best performance heuristics in the oracle(2) solving uniquely 26%

and 31% of the testsets on DS(1,2) respectively while NN heuristics are considered second and third best performance in oracle(3) solving 26% and 27% of test instances on DS(1,2) respectively. In terms of number of bins, many instances are solved equally using the heuristics in the pools however the new generated heuristics in oracles(2 and 3) are considered second and third best performance. Also, Extending the baseline oracle(1) to oracle(4) by adding the new heuristics, LSTM and NN heuristics solve very few instances equally and better than the original heuristics, i.e. 2 and 3 instances from DS1 and 2 respectively.

In the experiments using 20 bins, it is clear that the NN heuristics fail to handle larger numbers of bins (20), failing to solve any instance with performance better than the other methods. While LSTM heuristics solve 35 and 14 in DS1 and DS2 respectively from Falkenauer's performance perspective. In general, the new heuristics solve less instances in DS2 comparing to DS1. This is possibly due to a broader range of item sizes [20, 100] that are used to build DS2 instances (DS1 ranges from[40, 60]): the encoder in LSTM approach may have more difficulty summarising instances with a wide range of values, potentially requiring more encoder layers for the model.

As an example to show the instances that are solved equally best, figure 3 shows the number of instances that are solved using Oracle(4) heuristics on DS1 using 3 bins in terms of number of bins used. Each heuristic is a bubble with a size refers to the number of instances uniquely best solved by that particular heuristic. The width of the edges between the bubbles refers to the number of instances solved equally using the two related heuristics. As it is complicated to show the different combination of heuristics that solve the instances equally, we decompose the combinations into pairs, e.g. if an instance solved equally best using this set of heuristics (BF, FF and LSTM) then we count this instance in (LSTM, BF), (LSTM, FF) and (BF, FF) edges.

C. Comparing The Oracles

As we described in the introduction I, our aim is to improve the overall performance by adding new generated heuristics to a pool of human-designed heuristics. Assuming a perfect per-instance algorithm selector using three, four and five heuristics, i.e. Oracles 1, 2/3 and 4 respectively. Table VII shows the total performance from both Falkenauer's performance and number of bins perspectives on DS(1,2) using 3 and 20 bins. In general, adding LSTM-generated heuristic to the original oracle significantly improve the overall performance in the most cases while adding the NN-generated heuristic significantly improve the performance in the experiments using 3 bins only. Adding both the neural generated heuristics (LSTM and NN) leverage the complementary strength of both heuristics.

In order to exploit the performance complementarity of the different heuristics that cover different parts of the instance space, we create **Oracle(5)** that includes all the heuristics, i.e. the human-designed constructive and all the neural generated heuristics (10 LSTM heuristics and 10 NN heuristics). A pool with more well-designed heuristics is expected to achieve better results than pool with less number of heuristics. The results mainly back this up. Ultimately, the number of containers (bins) determines the cost of any real-world solution. Oracle(5) obtains the best performance improving over Oracle(1) and saving 38 and 53 bins in the experiments using 3 bins for DS(1 and 2) respectively. Using 20 bins, Oracle(5) uses 4 and 33 less bins than the baseline Oracle(1) on DS(1 and 2) respectively. Although saving 4 bins could be seen a small number of saving, those bins might be ships or planes and thus saving potential millions of dollars. Table VIII shows the results obtained from applying a Wilcoxon signed-rank test [35] with 5% confidence level to evaluate significance in a pairwise fashion of the new oracles to the baseline oracle. It is clear that adding more generated heuristics provides significant improvement in the most cases. It should be noted that Wilcoxon signed-rank test is a rank sum test(not median tests) and thus it is possible for the ranks to differ but the medians to be the same.

VII. Conclusion

We proposed a novel neural approach to generating constructive heuristics for dealing with streaming bin-packing problem that directly output decisions to pack items accounting for current bin states. Unlike typical methods to generate heuristics such as hyper-heuristics that search the heuristics space for potential good heuristics, our approach navigates the decision space for solving the problem instance. We test the approach by evaluating the contribution of the new generated heuristics to a pool created by combining them with a set of existing well-known heuristics in the bin-packing field with increasing number of available bins. We have used two datasets including long streams drawn from two different distributions. The results showed that the expanded pool with the best-of-run generated heuristics brings improvement in 26%-31% of cases in terms of Falkenauer's performance using small number of available bins. Also, since the different heuristics cover different parts of the instance space, adding all the generated heuristics leverages the performance complementarity strength and provides more improvement with saving up to 53 bins. Furthermore, as far as we are aware, it provides the first example of using a Encoder-Decoder network with long-short-term memory as a generative heuristic technique for streaming data with limited resources. Future work will extend the approach to deal with other dynamic streaming domains such as job-shop scheduling which has additional constraints.

References

[1] P. Kerschke, H. H. Hoos, F. Neumann, and H. Trautmann, "Automated algorithm selection: Survey and perspectives," *Evolutionary computation*, pp. 1–47, 2018.

[2] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward, "A classification of hyper-heuristic approaches: revisited," in *Handbook of Metaheuristics*. Springer, 2019, pp. 453–477.

[3] J. Branke, S. Nguyen, C. W. Pickardt, and M. Zhang, "Automated design of production scheduling heuristics: A review," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 1, pp. 110–124, 2015.

[4] N. Pillay and R. Qu, *Hyper-heuristics: theory and applications*. Springer, 2018.

TABLE VI

THE NUMBER OF INSTANCES IN TESTSETS FROM DS(1,2) THAT ARE UNIQUELY BEST SOLVED USING THE HEURISTICS PER ORACLE IN TERMS OF BOTH FALKENAUER'S PERFORMANCE AND NUMBER OF BINS USED.

DS1- 3 bins	Falkenauer's Performance						Number of Bins					
	BF	FF	WF	LSTM	NN	LSTM=NN	BF	FF	WF	LSTM	NN	LSTM=NN
Oracle(1)	50	50	50	—	—	—	42	7	16	—	—	—
Oracle(2)	44	27	31	47	—	—	33	7	8	14	—	—
Oracle(3)	49	39	20	—	41	—	41	7	8	—	7	—
Oracle(4)	43	22	15	37	31	0	33	7	7	12	4	2
DS1- 20 bins												
Oracle(1)	50	50	50	—	—	—	50	39	38	—	—	—
Oracle(2)	48	32	31	35	—	—	48	22	26	11	—	—
Oracle(3)	50	50	50	—	0	—	50	39	38	—	0	—
Oracle(4)	48	32	31	35	0	0	48	22	26	11	0	0
DS2- 3 bins												
Oracle(1)	50	50	50	—	—	—	43	16	6	—	—	—
Oracle(2)	48	40	22	40	—	—	39	11	5	11	—	—
Oracle(3)	49	35	27	—	39	—	38	12	5	—	7	—
Oracle(4)	47	33	15	31	24	0	35	9	4	8	3	3
DS2- 20 bins												
Oracle(1)	50	50	50	—	—	—	50	23	10	—	—	—
Oracle(2)	50	50	36	14	—	—	50	23	9	3	—	—
Oracle(3)	50	50	50	—	0	—	50	23	10	—	0	—
Oracle(4)	50	50	36	14	0	0	50	23	9	3	0	0

TABLE VII

THE TOTAL FALKENAUER'S PERFORMANCE AND NUMBER OF USED BINS PER ORACLE ON TESTSETS (150 INSTANCES) FROM DS(1,2) USING 3 AND 20 BINS, WHERE THE HIGHER THE FALKENAUER VALUE IS BETTER AND THE LOWER NUMBER OF BINS USED IS BETTER

	Falkenauer's Performance					Number of Bins				
	Oracle(1)	Oracle(2)	Oracle(3)	Oracle(4)	Oracle(5)	Oracle(1)	Oracle(2)	Oracle(3)	Oracle(4)	Oracle(5)
DS1- 3 bins	123.826	124.402	124.186	124.646	**125.376**	6639	6624	6631	6619	**6601**
DS1- 20 bins	134.153	134.706	134.153	134.706	**135.516**	6358	6345	6358	6345	**6325**
DS2- 3 bins	118.251	118.669	118.545	118.805	**119.932**	8255	8243	8248	8240	**8202**
DS2- 20 bins	134.488	134.636	134.488	134.636	**134.703**	7634	7631	7634	7631	**7630**

TABLE VIII

THE COMPARISON BETWEEN THE NEW ORACLES AND THE BASELINE ORACLE OVER THE DIFFERENT TESTSETS (150 INSTANCES) FROM DS(1,2) IN TERMS OF FALKENAUER'S PERFORMANCE AND NUMBER OF BINS USING WILCOXON SIGNED-RANK TEST. FOR A GIVEN PAIR OF ORACLE TESTS ORC.(A,B), THE ↑ MEANS THE FIRST ORACLE'S MEDIAN IS BETTER, ( ) MEANS BOTH ORACLES HAVE EQUAL MEDIAN, + MEANS THERE IS SIGNIFICANCE, MEANS THERE IS NO SIGNIFICANCE AND TYPICAL MEANS BOTH ORACLES ARE TYPICAL AND THE ADDED HEURISTIC FAILED TO OUTPERFORM THE CONSTRUCTIVE HEURISTICS.

	Falkenauer's Performance					Number of Bins				
DS- Bins	Orc.(2,1)	Orc.(3,1)	Orc.(4,1)	Orc.(2,3)	Orc.(5,1)	Orc.(2,1)	Orc.(3,1)	Orc.(4,1)	Orc.(2,3)	Orc.(5,1)
DS1- 3 Bins	↑ +	↑ +	↑ +	↑	↑ +	( ) +	( ) +	( ) +	( )	( ) +
DS1- 20 Bins	↑ +	Typical	↑ +	↑ +	↑ +	( ) +	Typical	( ) +	( ) +	( ) +
DS2- 3 Bins	↑ +	↑ +	↑ +	↑	↑ +	( ) +	( ) +	( ) +	( )	↑ +
DS2- 20 Bins	( ) +	Typical	( ) +	( ) +	( ) +	( )	Typical	( )	( )	( ) +

[5] J. R. Koza and J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.

[6] N. Pillay, "Evolving construction heuristics for the curriculum based university course timetabling problem," in *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2016, pp. 4437–4443.

[7] R. Raghavjee and N. Pillay, "The effect of construction heuristics on the performance of a genetic algorithm for the school timetabling problem," in *Proceedings of the South African Institute of Computer Scientists and Information Technologists Conference on Knowledge, Innovation and Leadership in a Diverse, Multidisciplinary Environment*, 2011, pp. 187–194.

[8] K. Sim and E. Hart, "A combined generative and selective hyper-heuristic for the vehicle routing problem," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 2016, pp. 1093–1100.

[9] R. Glanville, D. Griffiths, P. Baron, J. H. Drake, M. Hyde, K. Ibrahim, and E. Ozcan, "A genetic programming hyper-heuristic for the multidimensional knapsack problem," *Kybernetes*, 2014.

[10] A. Sosa-Ascencio, G. Ochoa, H. Terashima-Marin, and S. E. Conant-Pablos, "Grammar-based generation of variable-selection heuristics for constraint satisfaction problems," *Genetic Programming and Evolvable Machines*, vol. 17, no. 2, pp. 119–144, 2016.

[11] N. R. Sabar, M. Ayob, G. Kendall, and R. Qu, "Automatic design of a hyper-heuristic framework with gene expression programming for combinatorial optimization problems," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 3, pp. 309–325, 2014.

[12] ——, "A dynamic multiarmed bandit-gene expression programming hyper-heuristic for combinatorial optimization problems," *IEEE transactions on cybernetics*, vol. 45, no. 2, pp. 217–228, 2014.

[13] C. Stone, E. Hart, and B. Paechter, "Automatic generation of constructive heuristics for multiple types of combinatorial optimisation problems with grammatical evolution and geometric graphs," in *International Conference on the Applications of Evolutionary Computation*. Springer, 2018, pp. 578–593.

[14] J. H. Drake, N. Kililis, and E. Özcan, "Generation of vns components with grammatical evolution for vehicle routing," in *European conference on genetic programming*. Springer, 2013, pp. 25–36.